Sage Tutorial

Release 5.11

The Sage Group

15. 08. 2013

Inhaltsverzeichnis

Sage ist eine freie, Open-Source-Software, die Forschung und Lehre in Algebra, Geometrie, Zahlentheorie, Kryptographie, numerischen Berechnungen und verwandten Gebieten unterstützt. Sowohl das Entwicklungsmodell von Sage als auch die Technologie in Sage zeichnen sich durch eine extrem starke Betonung von Offenheit, Gemeinschaft, Kooperation und Zusammenarbeit aus. Wir bauen das Auto und erfinden nicht das Rad neu. Das Ziel von Sage ist es, eine aktiv gepflegte, freie Open-Source-Alternative zu Magma, Maple, Mathematica und Matlab zu entwickeln.

Dieses Tutorial ist die beste Möglichkeit mit Sage in wenigen Stunden vertraut zu werden. Sie können es im HTML- oder PDF-Format lesen oder im Sage-Notebook. Dazu klicken Sie zuerst auf `Help` und `Tutorial`, um innerhalb von Sage interaktiv mit dem Tutorial zu arbeiten.

KAPITEL 1

Einleitung

Um dieses Tutorial vollständig durchzuarbeiten sollten 3 bis 4 Stunden genügen. Sie können es im HTML oder PDF Format lesen oder im Sage Notebook. Dazu klicken Sie zuerst auf `Help` und `Tutorial`, um innerhalb von Sage interaktiv mit dem Tutorial zu arbeiten.

Obwohl große Teile von Sage mithilfe von Python implementiert sind, ist kein tieferes Verständnis von Python notwendig um dieses Tutorial lesen zu können. Sie werden Python zu einem gewissen Zeitpunkt lernen wollen (Python kann sehr viel Spass bereiten) und es gibt viele ausgezeichnete freie Quellen, wozu auch [PyT] und [Dive] gehören. Wenn Sie nur kurz etwas in Sage ausprobieren möchten, ist dieses Tutorial der richtige Ort um damit anzufangen. Zum Beispiel:

```
sage: 2 + 2
4
sage: factor(-2007)
-1 * 3^2 * 223

sage: A = matrix(4,4, range(16)); A
[ 0  1  2  3]
[ 4  5  6  7]
[ 8  9 10 11]
[12 13 14 15]

sage: factor(A.charpoly())
x^2 * (x^2 - 30*x - 80)

sage: m = matrix(ZZ,2, range(4))
sage: m[0,0] = m[0,0] - 3
sage: m
[-3  1]
[ 2  3]

sage: E = EllipticCurve([1,2,3,4,5]);
sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5
over Rational Field
sage: E.anlist(10)
[0, 1, 1, 0, -1, -3, 0, -1, -3, -3, -3]
sage: E.rank()
1
```

```
sage: k = 1/(sqrt(3)*I + 3/4 + sqrt(73)*5/9); k
36/(20*sqrt(73) + 36*I*sqrt(3) + 27)
sage: N(k)
0.165495678130644 - 0.0521492082074256*I
sage: N(k,30)      # 30 "bits"
0.16549568 - 0.052149208*I
sage: latex(k)
\frac{36}{20 \, \sqrt{73} + 36 i \, \sqrt{3} + 27}
```

1.1 Installation

Falls Sie Sage auf Ihrem Computer nicht installiert haben und nur ein paar Befehle ausführen möchten, können Sie es online unter http://www.sagenb.org benutzen.

Schauen Sie sich den Sage Installation Guide an, um Anleitungen zur Installation von Sage auf Ihrem Computer zu erhalten. Hier geben wir nur ein paar Kommentare ab.

1. Die herunterladbare Sage-Datei wurde nach der *batteries included* Philosophie zusammengestellt. In anderen Worten, obwohl Sage Python, IPython, PARI, GAP, Singular, Maxima, NTL, GMP und so weiter benutzt, müssen Sie diese Programme nicht separat installieren, da diese in der Sage-Distribution enthalten sind. Jedoch müssen Sie, um bestimmte Sage Zusatzfunktionen, zum Beispiel Macaulay oder KASH, nutzen zu können, diese entsprechenden optionalen Pakete installieren, oder zumindest die relevanten Programme auf ihrem Computer schon installiert haben. Macaulay und KASH sind Sage-Pakete (um eine Liste aller verfügbaren Sage-Pakete zu sehen, geben Sie `sage -optional` ein, oder rufen Sie die "Download" Seite auf der Sage Webseite auf).

2. Die vorkompilierte Binärversion von Sage (zu finden auf der Sage-Webseite) ist vielleicht einfacher und schneller zu installieren als die Quellcode-Version. Sie müssen die Datei nur entpacken und das Kommando `sage` ausführen.

3. Falls Sie das SageTeX-Paket benutzen möchten (mit welchem Sie die Ergebnisse von Sage Berechnungen in eine LaTeX-Datei einbauen können), müssen Sie SageTeX Ihrer TeX-Distribution bekannt machen. Um dies zu tun, lesen Sie den Abschnitt Make SageTeX known to TeX im Sage Installation Guide (Dieser Link sollte Sie zu eine lokalen Kopie des Installation Guides führen). Es ist ziemlich einfach; Sie müssen nur eine Umgebungsvariable setzen oder eine einzige Datei in ein Verzeichnis kopieren, welches TeX durchsucht.

   Die Dokumentation für SageTeX befindet sich in `$SAGE_ROOT/local/share/texmf/tex/generic/sagetex/`, wobei "`$SAGE_ROOT`" auf das Verzeichnis zeigt, in welches Sie Sage installiert haben, zum Beispiel `/opt/sage-4.2.1`.

1.2 Wie man Sage benutzen kann

Sie können Sage auf verschiedene Weise benutzen.

- **graphisches Notebook-Interface:** lesen Sie den Abschnitt zum Notebook im Referenzhandbuch und *Die Notebook Umgebung* weiter unten,
- **interaktive Kommandozeile:** lesen Sie *Die interaktive Kommandozeile*,
- **Programme:** Indem Sie interpretierte und kompilierte Programme in Sage schreiben (lesen Sie *Sage-Dateien Laden und Anhängen* und *Kompilierten Code erzeugen*), und
- **Skripte:** indem Sie eigenständige Pythonskripte schreiben, welche die Sage-Bibliothek benutzen (lesen Sie *eigenständige Python/Sage Skripte*).

1.3 Langfristige Ziele von Sage

- **nützlich**: Sages Zielgruppen sind Mathematikstudenten (von der Schule bis zur Universität), Lehrer und forschende Mathematiker. Das Ziel ist es, Software bereitzustellen, die benutzt werden kann, um mathematische Konstruktionen in der Algebra, Geometrie, Zahlentheorie, Analysis, Numerik, usw. zu erforschen und mit ihnen zu experimentieren. Sage hilft dabei, einfacher mit mathematischen Objekten experimentieren zu können.
- **effizient:** Schnell sein. Sage benutzt hochoptimierte ausgereifte Software wie GMP, PARI, GAP und NTL, und ist somit bei vielen Aufgaben sehr schnell.
- **frei und Open-Source:** Der Quellcode muss frei verfügbar und lesbar sein, damit Benutzer verstehen können, was das System gerade macht, und es einfacher erweitern zu können. Genauso wie Mathematiker ein tieferes Verständnis eines Theorems erlangen, indem sie den Beweis sorgfältig lesen oder zumindest überfliegen, sollten Leute, die Berechnungen durchführen, verstehen, wie die Berechnungen zustande kommen, indem sie den dokumentierten Quellcode lesen. Falls Sie Sage verwenden, um Berechnungen für ein Paper durchzuführen, welches Sie veröffentlichen, können Sie sicher sein, dass Ihre Leser immer freien Zugang zu Sage und seinem Quellcode haben und Sie dürfen sogar Ihre Sage Version archivieren und weiterverteilen.
- **einfach zu kompilieren:** Sage sollte für GNU/Linux, Mac OS X und Windowsbenutzer einfach aus dem Quellcode kompiliert werden können.
- **kooperativ** Sage stellt robuste Schnittstelle zu vielen anderen Computeralgebrasystemen, einschließlich PARI, GAP, Singular, Maxima, KASH, Magma, Maple und Mathematica zur Verfügung. Sage ist dazu gedacht, bestehende Mathematik-Software zu vereinheitlichen und zu erweitern.
- **gut dokumentiert:** Es gibt ein Tutorial, einen Programmierguide, ein Referenzhandbuch und Howtos mit zahlreichen Beispielen und Erläuterungen der dahinterstehenden Mathematik.
- **erweiterbar:** Es ist möglich, neue Datentypen zu definieren oder von eingebauten Typen abzuleiten und Code vieler verschiedener Sprachen zu benutzen.
- **benutzerfreundlich**: Es sollte einfach sein zu verstehen, welche Funktionalität für ein bestimmtes Objekt zur Verfügung gestellt wird und die Dokumentation und den Quellcode zu betrachten. Weiterhin sollte ein hochwertiger Benutzersupport erreicht werden.

KAPITEL 2

Eine begleitende Tour

Dieser Abschnitt ist eine begleitende Tour einiger in Sage vorhandener Funktionen. Um viele weitere Beispiele zu sehen, schauen Sie sich Sage Constructions an. Dies ist dazu gedacht, die allgemeine Frage "Wie konstruiere ich ... in Sage?" zu beantworten. Schauen Sie sich auch das Sage Reference Manual an, welches Tausende weiterer Beispiele beinhaltet. Beachten Sie auch, dass Sie diese Tour interaktiv im Sage-Notebook durcharbeiten können, indem Sie auf den `Help` Link klicken.

(Falls Sie dieses Tutorial im Sage-Notebook sehen, drücken Sie `shift-enter` um die Eingabe-Zellen auszuwerten. Sie können die Eingabe sogar verändern bevor Sie `shift-enter` drücken. Auf einigen Macs müssen Sie vielleicht `shift-return` anstelle von `shift-enter` drücken.)

2.1 Zuweisung, Gleichheit und Arithmetik

Bis auf wenige Ausnahmen benutzt Sage die Programmiersprache Python, deshalb werden Ihnen die meisten einführenden Bücher über Python dabei helfen Sage zu lernen.

Sage benutzt = für Zuweisungen und ==, <=, >=, < und > für Vergleiche.

```
sage: a = 5
sage: a
5
sage: 2 == 2
True
sage: 2 == 3
False
sage: 2 < 3
True
sage: a == 5
True
```

Sage unterstützt alle grundlegenden mathematischen Operationen:

```
sage: 2**3    #  ** bedeutet hoch
8
sage: 2^3     #  ^ ist ein Synonym für ** (anders als in Python)
8
sage: 10 % 3  #  für ganzzahlige Argumente bedeutet % mod, d.h. Restbildung
1
sage: 10/4
```

```
5/2
sage: 10//4   #  für ganzzahlige Argumente gibt // den \
....:         #  ganzzahligen Quotienten zurück
2
sage: 4 * (10 // 4) + 10 % 4 == 10
True
sage: 3^2*4 + 2%5
38
```

Die Berechnung eines Ausdrucks wie `3^2*4 + 2%5` hängt von der Reihenfolge ab, in der die Operationen ausgeführt werden. Dies wird in der "Operatorrangfolge-Tabelle" in *Binäre arithmetische Operatorrangfolge* festgelegt.

Sage stellt auch viele bekannte mathematische Funktionen zur Verfügung; hier sind nur ein paar Beispiele

```
sage: sqrt(3.4)
1.84390889145858
sage: sin(5.135)
-0.912021158525540
sage: sin(pi/3)
1/2*sqrt(3)
```

Wie das letzte Beispiel zeigt, geben manche mathematische Ausdrücke 'exakte' Werte anstelle von numerischen Approximationen zurück. Um eine numerische Approximation zu bekommen, können Sie entweder die Funktion `n` oder die Methode `n` verwenden (beide haben auch den längeren Namen, `numerical_approx`, und die Funktion `N` ist die gleiche wie `n`). Diese nehmen auch die optionalen Argumente `prec`, welches die gewünschte Anzahl von Bit an Genauigkeit ist und `digits`, welches die gewünschte Anzahl Dezimalstellen an Genauigkeit ist, entgegen; der Standardwert ist 53 Bit Genauigkeit.

```
sage: exp(2)
e^2
sage: n(exp(2))
7.38905609893065
sage: sqrt(pi).numerical_approx()
1.77245385090552
sage: sin(10).n(digits=5)
-0.54402
sage: N(sin(10),digits=10)
-0.5440211109
sage: numerical_approx(pi, prec=200)
3.1415926535897932384626433832795028841971693993751058209749
```

Python ist dynamisch typisiert, also ist dem Wert, auf den jede Variable weist, ein Typ zugeordnet; jedoch darf eine Variable Werte eines beliebigen Python-Typs innerhalb eines Sichtbarkeitsbereich aufnehmen.

```
sage: a = 5   # a ist eine ganze Zahl
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: a = 5/3  # jetzt ist a eine rationale Zahl
sage: type(a)
<type 'sage.rings.rational.Rational'>
sage: a = 'hello'  # jetzt ist a ein String
sage: type(a)
<type 'str'>
```

Die Programmiersprache C, welche statisch typisiert ist, unterscheidet sich hierzu stark; eine Variable, die dazu deklariert ist eine Ganzzahl (int) aufzunehmen, kann in ihrem Sichtbarkeitsbereich auch nur ganze Zahlen aufnehmen.

Für Verwirrung in Python sorgt häufig, dass Integer Literale, die mit Null beginnen als Oktalzahl, d.h. als Zahl zur Basis 8, behandelt werden.

```
sage: 011
9
sage: 8 + 1
9
sage: n = 011
sage: n.str(8)   # Darstellung von n als String zur Basis 8
'11'
```

Dies ist konsistent mit der Programmiersprache C.

2.2 Hilfe

Sage hat eine umfassende eingebaute Dokumentation, auf die zugegriffen werden kann, indem der Name der Funktion oder Konstanten (zum Beispiel) gefolgt von einem Fragezeichen eingegeben wird:

```
sage: tan?
Type:        <class 'sage.calculus.calculus.Function_tan'>
Definition:  tan( [noargspec] )
Docstring:

    The tangent function

    EXAMPLES:
        sage: tan(pi)
        0
        sage: tan(3.1415)
        -0.0000926535900581913
        sage: tan(3.1415/4)
        0.999953674278156
        sage: tan(pi/4)
        1
        sage: tan(1/2)
        tan(1/2)
        sage: RR(tan(1/2))
        0.546302489843790
sage: log2?
Type:        <class 'sage.functions.constants.Log2'>
Definition:  log2( [noargspec] )
Docstring:

    The natural logarithm of the real number 2.

    EXAMPLES:
        sage: log2
        log2
        sage: float(log2)
        0.69314718055994529
        sage: RR(log2)
        0.693147180559945
        sage: R = RealField(200); R
        Real Field with 200 bits of precision
        sage: R(log2)
        0.69314718055994530941723212145817656807550013436025525412068
        sage: l = (1-log2)/(1+log2); l
        (1 - log(2))/(log(2) + 1)
        sage: R(l)
```

```
        0.18123221829928249948761381864650311423330609774776013488056
        sage: maxima(log2)
        log(2)
        sage: maxima(log2).float()
        .6931471805599453
        sage: gp(log2)
        0.6931471805599453094172321215             # 32-bit
        0.69314718055994530941723212145817656807   # 64-bit
sage: sudoku?
File:        sage/local/lib/python2.5/site-packages/sage/games/sudoku.py
Type:        <type 'function'>
Definition:  sudoku(A)
Docstring:

    Solve the 9x9 Sudoku puzzle defined by the matrix A.

    EXAMPLE:
        sage: A = matrix(ZZ,9,[5,0,0, 0,8,0, 0,4,9, 0,0,0, 5,0,0,
    0,3,0, 0,6,7, 3,0,0, 0,0,1, 1,5,0, 0,0,0, 0,0,0, 0,0,0, 2,0,8, 0,0,0,
    0,0,0, 0,0,0, 0,1,8, 7,0,0, 0,0,4, 1,5,0,   0,3,0, 0,0,2,
    0,0,0, 4,9,0, 0,5,0, 0,0,3])
        sage: A
        [5 0 0 0 8 0 0 4 9]
        [0 0 0 5 0 0 0 3 0]
        [0 6 7 3 0 0 0 0 1]
        [1 5 0 0 0 0 0 0 0]
        [0 0 0 2 0 8 0 0 0]
        [0 0 0 0 0 0 0 1 8]
        [7 0 0 0 0 4 1 5 0]
        [0 3 0 0 0 2 0 0 0]
        [4 9 0 0 5 0 0 0 3]
        sage: sudoku(A)
        [5 1 3 6 8 7 2 4 9]
        [8 4 9 5 2 1 6 3 7]
        [2 6 7 3 4 9 5 8 1]
        [1 5 8 4 6 3 9 7 2]
        [9 7 4 2 1 8 3 6 5]
        [3 2 6 7 9 5 4 1 8]
        [7 8 2 9 3 4 1 5 6]
        [6 3 5 1 7 2 8 9 4]
        [4 9 1 8 5 6 7 2 3]
```

Sage stellt auch eine 'Tab-Vervollständigung' zur Verfügung: Schreiben Sie die ersten Buchstaben einer Funktion und drücken Sie danach die Tabulatortaste. Wenn Sie zum Beispiel `ta` gefolgt von `TAB` eingeben, wird Sage `tachyon, tan, tanh, taylor` ausgeben. Dies ist eine gute Möglichkeit den Namen von Funktionen und anderen Strukturen in Sage herauszufinden.

2.3 Funktionen, Einrückungen, und Zählen

Um in Sage eine neue Funktion zu definieren, können Sie den `def` Befehl und einen Doppelpunkt nach der Liste der Variablennamen benutzen. Zum Beispiel:

```
sage: def is_even(n):
...       return n%2 == 0
...
sage: is_even(2)
```

```
True
sage: is_even(3)
False
```

Anmerkung: Abhängig von der Version des Tutorials, das Sie gerade lesen, sehen Sie vielleicht drei Punkte `...` in der zweiten Zeile dieses Beispiels. Tippen Sie diese nicht; sie sind nur da um zu verdeutlichen, dass der Code eingerückt ist. Wann immer dies der Fall ist, drücken Sie [Return/Enter] einmal am Ende des Blocks um eine Leerzeile einzufügen und die Funktionsdefinition zu beenden.

Sie bestimmen den Typ ihrer Eingabeargumente nicht. Sie können mehrere Argumente festlegen, jedes davon kann einen optionalen Standardwert haben. Zum Beispiel wird in der Funktion unterhalb standardmäßig der Wert `divisor=2` benutzt, falls `divisor` nicht angegeben wurde.

```
sage: def is_divisible_by(number, divisor=2):
...       return number%divisor == 0
sage: is_divisible_by(6,2)
True
sage: is_divisible_by(6)
True
sage: is_divisible_by(6, 5)
False
```

Sie können auch ein oder mehrere Eingabeargumente explizit angeben wenn Sie die Funktion aufrufen; wenn Sie die Eingaben explizit angeben, können Sie dies in beliebiger Reihenfolge tun:

```
sage: is_divisible_by(6, divisor=5)
False
sage: is_divisible_by(divisor=2, number=6)
True
```

In Python werden Codeblöcke nicht mit geschweiften Klammern oder "begin-" und "end-Blöcken" kenntlich gemacht. Stattdessen werden Codeblöcke durch Einrückungen bestimmt, welche exakt zusammenpassen müssen. Zum Beispiel ist das Folgende ein Syntaxfehler, da die `return` Anweisung nicht genauso weit eingerückt ist wie die anderen Zeilen zuvor.

```
sage: def even(n):
...       v = []
...       for i in range(3,n):
...           if i % 2 == 0:
...               v.append(i)
...      return v
Syntax Error:
       return v
```

Wenn Sie die Einrückung korrigieren, funktioniert die Funktion:

```
sage: def even(n):
...       v = []
...       for i in range(3,n):
...           if i % 2 == 0:
...               v.append(i)
...       return v
sage: even(10)
[4, 6, 8]
```

Semikola sind an den Zeilenenden nicht notwendig; sie können jedoch mehrere Anweisungen, mit Semikola getrennt, in eine Zeile schreiben:

```
sage: a = 5; b = a + 3; c = b^2; c
64
```

Falls Sie möchten, dass sich eine einzelne Codezeile über mehrere Zeilen erstreckt, können Sie einen terminierenden Backslash verwenden:

```
sage: 2 + \
...      3
5
```

In Sage können Sie zählen indem Sie über einen Zahlenbereich iterieren. Zum Beispiel ist nächste Zeile unterhalb gleichwertig zu `for(i=0; i<3; i++)` in C++ oder Java:

```
sage: for i in range(3):
...       print i
0
1
2
```

Die nächste Zeile unterhalb ist gleichwertig zu `for(i=2;i<5;i++)`.

```
sage: for i in range(2,5):
...       print i
2
3
4
```

Das dritte Argument bestimmt die Schrittweite, also ist das Folgende gleichwertig zu `for(i=1;i<6;i+=2)`.

```
sage: for i in range(1,6,2):
...       print i
1
3
5
```

Oft will man eine schöne Tabelle erstellen, um die mit Sage berechneten Zahlen auszugeben. Eine einfache Möglichkeit dies zu tun ist String-Formatierung zu verwenden. Unten erstellen wir drei Spalten, jede genau 6 Zeichen breit, und erzeugen somit eine Tabelle mit Quadrat- und Kubikzahlen.

```
sage: for i in range(5):
...       print '%6s %6s %6s'%(i, i^2, i^3)
     0      0      0
     1      1      1
     2      4      8
     3      9     27
     4     16     64
```

Die elementarste Datenstruktur in Sage ist die Liste. Sie ist – wie der Name schon sagt – nichts anderes als eine Liste beliebiger Objekte. Zum Beispiel erzeugt der `range` Befehl, den wir schon verwendet haben, eine Liste:

```
sage: range(2,10)
[2, 3, 4, 5, 6, 7, 8, 9]
```

Hier ist eine etwas kompliziertere Liste:

```
sage: v = [1, "hello", 2/3, sin(x^3)]
sage: v
[1, 'hello', 2/3, sin(x^3)]
```

Listenindizierung beginnt, wie in vielen Programmiersprachen, bei 0.

```
sage: v[0]
1
sage: v[3]
sin(x^3)
```

Benutzen Sie `len(v)` um die Länge von `v` zu erhalten, benutzen Sie `v.append(obj)` um ein neues Objekt an das Ende von `v` anzuhängen, und benutzen Sie `del v[i]` um den i^{ten} Eintrag von `v` zu löschen:

```
sage: len(v)
4
sage: v.append(1.5)
sage: v
[1, 'hello', 2/3, sin(x^3), 1.50000000000000]
sage: del v[1]
sage: v
[1, 2/3, sin(x^3), 1.50000000000000]
```

Eine weitere wichtige Datenstruktur ist das Dictionary (oder assoziatives Array). Dies funktioniert wie eine Liste, außer dass es mit fast jedem Objekt indiziert werden kann (die Indizes müssen jedoch unveränderbar sein):

```
sage: d = {'hi':-2,  3/8:pi,   e:pi}
sage: d['hi']
-2
sage: d[e]
pi
```

Sie können auch neue Datentypen definieren, indem Sie Klassen verwenden. Mathematische Objekte mit Klassen zusammenzufassen ist eine mächtige Technik, die dabei helfen kann Sage-Programme zu vereinfachen und zu organisieren. Unten definieren wir eine Klasse, welche die Liste der geraden Zahlen bis *n* darstellt; Sie wird von dem Standard-Typ `list` abgeleitet.

```
sage: class Evens(list):
...       def __init__(self, n):
...           self.n = n
...           list.__init__(self, range(2, n+1, 2))
...       def __repr__(self):
...           return "Even positive numbers up to n."
```

Die `__init__` Methode wird aufgerufen um das Objekt zu initialisieren, wenn es erzeugt wird; die `__repr__` Method gibt einen Objekt-String aus. Wir rufen die Listen-Konstruktor-Methode in der zweite Zeile der `__init__` Methode. Ein Objekt der Klasse `Evens` erzeugen wir wie folgt:

```
sage: e = Evens(10)
sage: e
Even positive numbers up to n.
```

Beachten Sie, dass die Ausgabe von `e` die `__repr__` Methode verwendet, die wir definiert haben. Um die eigentliche Liste sehen zu können, benutzen wir die `list`-Funktion:

```
sage: list(e)
[2, 4, 6, 8, 10]
```

Wir können auch das `n` Attribut verwenden oder `e` wie eine Liste behandeln.

```
sage: e.n
10
sage: e[2]
6
```

2.4 Elementare Algebra und Analysis

Sage kann viele zur elementaren Algebra und Analysis gehörende Probleme lösen. Zum Beispiel: Lösungen von Gleichungen finden, Differentiation, Integration, und Laplace-Transformationen berechnen. Lesen Sie die Sage Constructions Dokumentation um weitere Beispiele zu finden.

2.4.1 Lösen von Gleichungen

Gleichungen exakt lösen

Die `solve` Funktion löst Gleichungen. Legen Sie zunächst Variablen an, bevor Sie diese benutzen; Die Argumente von `solve` sind eine Gleichung (oder ein System von Gleichungen) zusammen mit den Variablen, nach welchen Sie auflösen möchten:

```
sage: x = var('x')
sage: solve(x^2 + 3*x + 2, x)
[x == -2, x == -1]
```

Sie können eine Gleichung nach einer Variablen, in Abhängigkeit von den anderen, auflösen:

```
sage: x, b, c = var('x b c')
sage: solve([x^2 + b*x + c == 0],x)
[x == -1/2*b - 1/2*sqrt(b^2 - 4*c), x == -1/2*b + 1/2*sqrt(b^2 - 4*c)]
```

Sie können auch nach mehreren Variablen auflösen:

```
sage: x, y = var('x, y')
sage: solve([x+y==6, x-y==4], x, y)
[[x == 5, y == 1]]
```

Das folgende Beispiel, in dem Sage benutzt wird um ein System von nichtlinearen Gleichungen zu lösen, stammt von Jason Grout. Zunächst lösen wir das System symbolisch:

```
sage: var('x y p q')
(x, y, p, q)
sage: eq1 = p+q==9
sage: eq2 = q*y+p*x==-6
sage: eq3 = q*y^2+p*x^2==24
sage: solve([eq1,eq2,eq3,p==1],p,q,x,y)
[[p == 1, q == 8, x == -4/3*sqrt(10) - 2/3, y == 1/6*sqrt(5)*sqrt(2) - 2/3],
 [p == 1, q == 8, x == 4/3*sqrt(10) - 2/3, y == -1/6*sqrt(5)*sqrt(2) - 2/3]]
```

Um eine numerische Approximation der Lösungen zu erhalten können Sie stattdessen wie folgt vorgehen:

```
sage: solns = solve([eq1,eq2,eq3,p==1],p,q,x,y, solution_dict=True)
sage: [[s[p].n(30), s[q].n(30), s[x].n(30), s[y].n(30)] for s in solns]
[[1.0000000, 8.0000000, -4.8830369, -0.13962039],
 [1.0000000, 8.0000000, 3.5497035, -1.1937129]]
```

(Die Funktion `n` gibt eine numerische Approximation zurück, ihr Argument ist die Anzahl der Bits an Genauigkeit.)

Gleichungen numerisch lösen

Oftmals kann `solve` keine exakte Lösung der angegebenen Gleichung bzw. Gleichungen finden. Wenn dies passiert können Sie `find_root` verwenden um eine numerische Approximation zu finden. Beispielsweise gibt `solve` bei folgender Gleichung nichts brauchbares zurück:

```
sage: theta = var('theta')
sage: solve(cos(theta)==sin(theta), theta)
[sin(theta) == cos(theta)]
```

Wir können jedoch `find_root` verwenden um eine Lösung der obigen Gleichung im Bereich $0 < \phi < \pi/2$ zu finden:

```
sage: phi = var('phi')
sage: find_root(cos(phi)==sin(phi),0,pi/2)
0.785398163397448...
```

2.4.2 Differentiation, Integration, etc.

Sage weiß wie man viele Funktionen differenziert und integriert. Zum Beispiel können Sie folgendes eingeben um $\sin(u)$ nach u abzuleiten:

```
sage: u = var('u')
sage: diff(sin(u), u)
cos(u)
```

Um die vierte Ableitung $\sin(x^2)$ zu berechnen:

```
sage: diff(sin(x^2), x, 4)
16*x^4*sin(x^2) - 48*x^2*cos(x^2) - 12*sin(x^2)
```

Um die partiellen Ableitungen von $x^2 + 17y^2$ nach x beziehungsweise y zu berechnen:

```
sage: x, y = var('x,y')
sage: f = x^2 + 17*y^2
sage: f.diff(x)
2*x
sage: f.diff(y)
34*y
```

Wir machen weiter mit Integralen, sowohl bestimmt als auch unbestimmt. Die Berechnung von $\int x\sin(x^2)\,dx$ und $\int_0^1 \frac{x}{x^2+1}\,dx$:

```
sage: integral(x*sin(x^2), x)
-1/2*cos(x^2)
sage: integral(x/(x^2+1), x, 0, 1)
1/2*log(2)
```

Die Partialbruchzerlegung von $\frac{1}{x^2-1}$:

```
sage: f = 1/((1+x)*(x-1))
sage: f.partial_fraction(x)
-1/2/(x + 1) + 1/2/(x - 1)
```

2.4.3 Lösen von Differentialgleichungen

Sie können Sage verwenden um gewöhnliche Differentialgleichungen zu berechnen. Die Gleichung $x' + x - 1 = 0$ berechnen Sie wie folgt:

```
sage: t = var('t')    # definiere die Variable t
sage: x = function('x',t)   # definiere x als Funktion dieser Variablen
sage: DE = diff(x, t) + x - 1
```

```
sage: desolve(DE, [x,t])
(c + e^t)*e^(-t)
```

Dies benutzt Sages Schnittstelle zu Maxima [Max], daher kann sich die Ausgabe ein wenig von anderen Ausgaben in Sage unterscheiden. In diesem Fall wird mitgeteilt, dass $x(t) = e^{-t}(e^t + c)$ die allgemeine Lösung der Differentialgleichung ist.

Sie können auch Laplace-Transformationen berechnen: Die Laplace-Transformation von $t^2 e^t - \sin(t)$ wird wie folgt berechnet:

```
sage: s = var("s")
sage: t = var("t")
sage: f = t^2*exp(t) - sin(t)
sage: f.laplace(t,s)
-1/(s^2 + 1) + 2/(s - 1)^3
```

Hier ist ein komplizierteres Beispiel. Die Verschiebung des Gleichgewichts einer verkoppelten Feder, die an der linken Wand befestigt ist,

```
|------\/\/\/\/\---|Masse1|----\/\/\/\/\/----|Masse2|
         Feder1                   Feder2
```

wird durch dieses System der Differentialgleichungen zweiter Ordnung modelliert,

$$m_1 x_1'' + (k_1 + k_2)x_1 - k_2 x_2 = 0$$
$$m_2 x_2'' + k_2(x_2 - x_1) = 0,$$

wobei m_i die Masse des Objekts *i*, x_i die Verschiebung des Gleichgewichts der Masse *i* und k_i die Federkonstante der Feder *i* ist.

Beispiel: Benutzen Sie Sage um das obige Problem mit folgenden Werten zu lösen: $m_1 = 2$, $m_2 = 1$, $k_1 = 4$, $k_2 = 2$, $x_1(0) = 3$, $x_1'(0) = 0$, $x_2(0) = 3$, $x_2'(0) = 0$.

Lösung: Berechnen Sie die Laplace-Transformierte der ersten Gleichung (mit der Notation $x = x_1$, $y = x_2$):

```
sage: de1 = maxima("2*diff(x(t),t, 2) + 6*x(t) - 2*y(t)")
sage: lde1 = de1.laplace("t","s"); lde1
2*(-?%at('diff(x(t),t,1),t=0)+s^2*'laplace(x(t),t,s)-x(0)*s)-2*'laplace(y(t),t,s)+6*'laplace(x(t),t,s
```

Das ist schwierig zu lesen, es besagt jedoch, dass

$$-2x'(0) + 2s^2 \cdot X(s) - 2sx(0) - 2Y(s) + 6X(s) = 0$$

(wobei die Laplace-Transformierte der Funktion mit kleinem Anfangsbuchstaben $x(t)$ die Funktion mit großem Anfangsbuchstaben $X(s)$ ist). Berechnen Sie die Laplace-Transformierte der zweiten Gleichung:

```
sage: de2 = maxima("diff(y(t),t, 2) + 2*y(t) - 2*x(t)")
sage: lde2 = de2.laplace("t","s"); lde2
-?%at('diff(y(t),t,1),t=0)+s^2*'laplace(y(t),t,s)+2*'laplace(y(t),t,s)-2*'laplace(x(t),t,s)-y(0)*s
```

Dies besagt

$$-Y'(0) + s^2 Y(s) + 2Y(s) - 2X(s) - sy(0) = 0.$$

Setzen Sie die Anfangsbedingungen für $x(0)$, $x'(0)$, $y(0)$ und $y'(0)$ ein, und lösen die beiden Gleichungen, die Sie so erhalten:

```
sage: var('s X Y')
(s, X, Y)
sage: eqns = [(2*s^2+6)*X-2*Y == 6*s, -2*X +(s^2+2)*Y == 3*s]
sage: solve(eqns, X,Y)
[[X == 3*(s^3 + 3*s)/(s^4 + 5*s^2 + 4),
  Y == 3*(s^3 + 5*s)/(s^4 + 5*s^2 + 4)]]
```

Berechnen Sie jetzt die inverse Laplace-Transformierte um die Antwort zu erhalten:

```
sage: var('s t')
(s, t)
sage: inverse_laplace((3*s^3 + 9*s)/(s^4 + 5*s^2 + 4),s,t)
cos(2*t) + 2*cos(t)
sage: inverse_laplace((3*s^3 + 15*s)/(s^4 + 5*s^2 + 4),s,t)
-cos(2*t) + 4*cos(t)
```

Also ist die Lösung:

$$x_1(t) = \cos(2t) + 2\cos(t), \quad x_2(t) = 4\cos(t) - \cos(2t).$$

Die kann folgenderweise parametrisiert geplottet werden:

```
sage: t = var('t')
sage: P = parametric_plot((cos(2*t) + 2*cos(t), 4*cos(t) - cos(2*t) ),\
...    (t, 0, 2*pi), rgbcolor=hue(0.9))
sage: show(P)
```

Die einzelnen Komponenten können so geplottet werden:

```
sage: t = var('t')
sage: p1 = plot(cos(2*t) + 2*cos(t), (t,0, 2*pi), rgbcolor=hue(0.3))
sage: p2 = plot(4*cos(t) - cos(2*t), (t,0, 2*pi), rgbcolor=hue(0.6))
sage: show(p1 + p2)
```

Um mehr über das Plotten zu erfahren lesen Sie *Plotten*. Lesen Sie Abschnitt 5.5 von [NagleEtAl2004] um weitere Informationen über Differentialgleichungen zu erhalten.

2.4.4 Das Euler-Verfahren zur Lösung von Systemen von Differentialgleichungen

Im nächsten Beispiel illustrieren wir das Euler-Verfahren für ODEs erster und zweiter Ordnung. Wir rufen zunächst die grundlegende Idee für Differentialgleichungen erster Ordnung in Erinnerung. Sei ein Anfangswertproblem der Form

$$y' = f(x, y), \quad y(a) = c,$$

gegeben. Wir möchten eine Approximation des Wertes der Lösung bei $x = b$ mit $b > a$ finden.

Machen Sie sich anhand der Definition der Ableitung klar, dass

$$y'(x) \approx \frac{y(x+h) - y(x)}{h},$$

wobei $h > 0$ vorgegeben und klein ist. Zusammen mit der Differentialgleichung gibt dies $f(x, y(x)) \approx \frac{y(x+h)-y(x)}{h}$. Jetzt lösen wir nach $y(x + h)$ auf:

$$y(x + h) \approx y(x) + h \cdot f(x, y(x)).$$

Wenn wir $h \cdot f(x, y(x))$ den "Korrekturterm", $y(x)$ den "alten Wert von y" und $y(x+h)$ den "neuen Wert von y" nennen, kann diese Approximation neu ausgedrückt werden als:

$$y_{new} \approx y_{old} + h \cdot f(x, y_{old}).$$

Wenn wir das Intervall von a bis b in n Teilintervalle aufteilen, so dass $h = \frac{b-a}{n}$ gilt, können wir die Information in folgender Tabelle festhalten.

x	y	$h \cdot f(x,y)$
a	c	$h \cdot f(a,c)$
$a+h$	$c + h \cdot f(a,c)$	...
$a+2h$	...	
...		
$b = a + nh$	???	...

Unser Ziel ist zeilenweise alle leeren Einträge der Tabelle auszufüllen, bis wir den Eintrag ??? erreichen, welcher die Approximation des Euler-Verfahrens für $y(b)$ ist.

Die Idee für Systeme von ODEs ist ähnlich.

Beispiel: Approximiere $z(t)$, mit 4 Schritten der Eulermethode numerisch bei $t = 1$, wobei $z'' + tz' + z = 0$, $z(0) = 1$ und $z'(0) = 0$ ist.

Wir müssen die ODE zweiter Ordnung auf ein System von zwei Differentialgleichungen erster Ordnung reduzieren (wobei $x = z$, $y = z'$) und das Euler-Verfahren anwenden:

```
sage: t,x,y = PolynomialRing(RealField(10),3,"txy").gens()
sage: f = y; g = -x - y * t
sage: eulers_method_2x2(f,g, 0, 1, 0, 1/4, 1)
      t                x            h*f(t,x,y)                y       h*g(t,x,y)
      0                1                  0.00                0            -0.25
    1/4              1.0                -0.062            -0.25            -0.23
    1/2             0.94                 -0.12            -0.48            -0.17
    3/4             0.82                 -0.16            -0.66           -0.081
      1             0.65                 -0.18            -0.74            0.022
```

Also ist $z(1) \approx 0.75$.

Wir können auch die Punkte (x, y) plotten um ein ungefähres Bild der Kurve zu erhalten. Die Funktion `eulers_method_2x2_plot` macht dies; um sie zu benutzen, müssen wir die Funktionen f und g definieren, welche ein Argument mit drei Koordinaten (t, x, y) erwarten.

```
sage: f = lambda z: z[2]         # f(t,x,y) = y
sage: g = lambda z: -sin(z[1])   # g(t,x,y) = -sin(x)
sage: P = eulers_method_2x2_plot(f,g, 0.0, 0.75, 0.0, 0.1, 1.0)
```

Zu diesem Zeitpunkt enthält `P` die beiden Plots `P[0]` (der Plot von x nach t) und `P[1]` (der Plot von y nach t). Wir können beide wie folgt anzeigen:

```
sage: show(P[0] + P[1])
```

(Um mehr über das Plotten zu erfahren, lesen Sie *Plotten*.)

2.4.5 Spezielle Funktionen

Mehrere orthogonale Polynome und spezielle Funktionen sind implementiert, wobei sowohl PARI [GP] als auch Maxima [Max] verwendet wird. Sie sind in den dazugehörigen Abschnitten ("Orthogonal polynomials" beziehungsweise "Special functions") des Sage Referenzhandbuchs dokumentiert.

```
sage: x = polygen(QQ, 'x')
sage: chebyshev_U(2,x)
4*x^2 - 1
sage: bessel_I(1,1).n(250)
0.56515910399248502720769602760986330732889962162109200948029448947925564096
sage: bessel_I(1,1).n()
0.565159103992485
sage: bessel_I(2,1.1).n()
0.167089499251049
```

Zum jetzigen Zeitpunkt, enthält Sage nur Wrapper-Funktionen für numerische Berechnungen. Um symbolisch zu rechen, rufen Sie die Maxima-Schnittstelle bitte, wie im folgenden Beispiel, direkt auf

```
sage: maxima.eval("f:bessel_y(v, w)")
'bessel_y(v,w)'
sage: maxima.eval("diff(f,w)")
'(bessel_y(v-1,w)-bessel_y(v+1,w))/2'
```

2.5 Plotten

Sage kann zwei- und dreidimensionale Plots erzeugen.

2.5.1 Zweidimensionale Plots

Sage kann in zwei Dimensionen Kreise, Linien und Polygone zeichnen, sowie Plots von Funktionen in kartesischen Koordinaten und Plots in Polarkoordinaten, Konturplots und Plots von Vektorfeldern. Wir geben davon im Folgenden einige Beispiele an. Für weitere Beispiele zum Plotten mit Sage lesen Sie *Lösen von Differentialgleichungen* und *Maxima*, sowie die Sage Constructions Dokumentation.

Dieser Befehl erstellt einen gelben Kreis vom Radius 1 mit dem Ursprung als Zentrum:

```
sage: circle((0,0), 1, rgbcolor=(1,1,0))
```

Sie können auch einen ausgefüllten Kreis erzeugen:

```
sage: circle((0,0), 1, rgbcolor=(1,1,0), fill=True)
```

Sie können einen Kreis auch erstellen, indem Sie ihn einer Variable zuweisen; so wird kein Plot gezeigt.

```
sage: c = circle((0,0), 1, rgbcolor=(1,1,0))
```

Um den Plot zu zeigen, benutzen Sie `c.show()` oder `show(c)` wie folgt:

```
sage: c.show()
```

Alternativ führt das Auswerten von `c.save('filename.png')` dazu, dass der Plot in der angegebenen Datei gespeichert wird.

Noch sehen diese 'Kreise' jedoch eher wie Ellipsen aus, da die Achsen unterschiedlich skaliert sind. Sie können dies korrigieren:

```
sage: c.show(aspect_ratio=1)
```

Der Befehl `show(c, aspect_ratio=1)` erreicht das Gleiche. Sie können das Bild auch speichern, indem Sie `c.save('filename.png', aspect_ratio=1)` benutzen.

Es ist einfach elementare Funktionen zu plotten:

```
sage: plot(cos, (-5,5))
```

Sobald Sie einen Variablennamen angegeben haben, können Sie parametrische Plots erzeugen:

```
sage: x = var('x')
sage: parametric_plot((cos(x),sin(x)^3),(x,0,2*pi),rgbcolor=hue(0.6))
```

Es ist wichtig zu beachten, dass sich die Achsen eines Plots nur schneiden, wenn sich der Ursprung im angezeigten Bildbereich des Graphen befindet und ab einer bestimmten Größe der Werte wird die wissenschaftliche Notation benutzt:

```
sage: plot(x^2,(x,300,500))
```

Sie können mehrere Plots zusammenfügen indem Sie diese addieren:

```
sage: x = var('x')
sage: p1 = parametric_plot((cos(x),sin(x)),(x,0,2*pi),rgbcolor=hue(0.2))
sage: p2 = parametric_plot((cos(x),sin(x)^2),(x,0,2*pi),rgbcolor=hue(0.4))
sage: p3 = parametric_plot((cos(x),sin(x)^3),(x,0,2*pi),rgbcolor=hue(0.6))
sage: show(p1+p2+p3, axes=false)
```

Eine gute Möglichkeit ausgefüllte Figuren zu erstellen ist, eine Liste von Punkten zu erzeugen (`L` im folgenden Beispiel) und dann den `polygon` Befehl zu verwenden um die Figur mit dem, durch die Punkte bestimmten, Rand zu zeichnen. Zum Beispiel ist hier ein grünes Deltoid:

```
sage: L = [[-1+cos(pi*i/100)*(1+cos(pi*i/100)),\
...    2*sin(pi*i/100)*(1-cos(pi*i/100))] for i in range(200)]
sage: p = polygon(L, rgbcolor=(1/8,3/4,1/2))
sage: p
```

Geben Sie `show(p, axes=false)` ein, um dies ohne Achsen zu sehen.

Sie können auch Text zu einem Plot hinzufügen:

```
sage: L = [[6*cos(pi*i/100)+5*cos((6/2)*pi*i/100),\
...    6*sin(pi*i/100)-5*sin((6/2)*pi*i/100)] for i in range(200)]
sage: p = polygon(L, rgbcolor=(1/8,1/4,1/2))
sage: t = text("hypotrochoid", (5,4), rgbcolor=(1,0,0))
sage: show(p+t)
```

Analysis Lehrer zeichnen häufig den folgenden Plot an die Tafel: nicht nur einen Zweig von der arcsin Funktion, sondern mehrere, also den Plot von $y = \sin(x)$ für x zwischen -2π und 2π, an der 45 Grad Linie gespiegelt. Der folgende Sage Befehl erzeugt dies:

```
sage: v = [(sin(x),x) for x in srange(-2*float(pi),2*float(pi),0.1)]
sage: line(v)
```

Da die Tangensfunktion einen größeren Wertebereich als die Sinusfunktion besitzt, sollten Sie, falls Sie den gleichen Trick anwenden um die Inverse der Tangensfunktion zu plotten, das Minimum und Maximum der Koordinaten für die *x*-Achse ändern:

```
sage: v = [(tan(x),x) for x in srange(-2*float(pi),2*float(pi),0.01)]
sage: show(line(v), xmin=-20, xmax=20)
```

Sage berechnet auch Plots in Polarkoordinaten, Konturplots, Plots von Vektorfeldern (für besondere Arten von Funktionen). Hier ist ein Beispiel eines Konturplots:

```
sage: f = lambda x,y: cos(x*y)
sage: contour_plot(f, (-4, 4), (-4, 4))
```

2.5.2 Dreidimensionale Plots

Sage kann auch dazu verwendet werden dreidimensionale Plots zu zeichnen. Sowohl im Notebook, als auch von der Kommandozeile aus werden diese Plots standardmäßig mit den Open-Source-Paket [Jmol] angezeigt, welches interaktives Drehen und Zoomen der Grafik mit Hilfe der Maus unterstützt.

Benutzen Sie `plot3d` um eine Funktion der Form $f(x, y) = z$ zu zeichnen:

```
sage: x, y = var('x,y')
sage: plot3d(x^2 + y^2, (x,-2,2), (y,-2,2))
```

Alternativ können Sie auch `parametric_plot3d` verwenden um eine parametrisierte Fläche zu zeichnen, wobei jede der Variablen x, y, z durch eine Funktion einer oder zweier Variablen bestimmt ist. (Die Argumente sind typischerweise u und v). Der vorherige Plot kann wie folgt parametrisiert angegeben werden:

```
sage: u, v = var('u, v')
sage: f_x(u, v) = u
sage: f_y(u, v) = v
sage: f_z(u, v) = u^2 + v^2
sage: parametric_plot3d([f_x, f_y, f_z], (u, -2, 2), (v, -2, 2))
```

Die dritte Möglichkeit eine 3D Oberfläche zuplotten ist `implicit_plot3d`, dies zeichnet eine Kontur einer Funktion mit $f(x, y, z) = 0$ (so wird eine Punktmenge definiert). Wir können die Sphäre mithilfe einer klassischen Formel zeichnen:

```
sage: x, y, z = var('x, y, z')
sage: implicit_plot3d(x^2 + y^2 + z^2 - 4, (x,-2, 2), (y,-2, 2), (z,-2, 2))
```

Hier sind noch ein paar Beispiele:

Whitneys Regenschirm:

```
sage: u, v = var('u,v')
sage: fx = u*v
sage: fy = u
sage: fz = v^2
sage: parametric_plot3d([fx, fy, fz], (u, -1, 1), (v, -1, 1),
...   frame=False, color="yellow")
```

Die Kreuz-Kappe:

```
sage: u, v = var('u,v')
sage: fx = (1+cos(v))*cos(u)
sage: fy = (1+cos(v))*sin(u)
sage: fz = -tanh((2/3)*(u-pi))*sin(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi),
...   frame=False, color="red")
```

Ein gedrehter Torus:

```
sage: u, v = var('u,v')
sage: fx = (3+sin(v)+cos(u))*cos(2*v)
sage: fy = (3+sin(v)+cos(u))*sin(2*v)
sage: fz = sin(u)+2*cos(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi),
...   frame=False, color="red")
```

Die Lemniskate:

```
sage: x, y, z = var('x,y,z')
sage: f(x, y, z) = 4*x^2 * (x^2 + y^2 + z^2 + z) + y^2 * (y^2 + z^2 - 1)
sage: implicit_plot3d(f, (x, -0.5, 0.5), (y, -1, 1), (z, -1, 1))
```

s.. _section-functions-issues:

2.6 Häufige Probleme mit Funktionen

Das Definieren von Funktionen kann in mancher Hinsicht verwirrend sein (z.B. beim Ableiten oder Plotten). In diesem Abschnitt versuchen wir die relevanten Probleme anzusprechen.

Nun erläutern wir verschiedene Möglichkeiten Dinge zu definieren, die das Recht haben könnten "Funktionen" genannt zu werden:

1. Definition einer Python-Funktion wie in *Funktionen, Einrückungen, und Zählen* beschrieben. Diese Funktionen können geplottet, aber nicht abgeleitet oder integriert werden.

```
sage: def f(z): return z^2
sage: type(f)
<type 'function'>
sage: f(3)
9
sage: plot(f, 0, 2)
```

Beachten Sie die Syntax in der letzten Zeile. Falls Sie stattdessen `plot(f(z), 0, 2)` verwenden, erhalten Sie einen Fehler, da `z` eine Dummy-Variable in der Definition von `f` ist, und außerhalb dieser nicht definiert ist. In der Tat gibt sogar nur `f(z)` einen Fehler zurück. Das Folgende funktioniert in diesem Fall, obwohl es im Allgemeinen Probleme verursachen kann und deshalb vermieden werden sollte. (Beachten Sie unten den 4. Punkt)

```
sage: var('z')   # z wird als Variable definiert
z
sage: f(z)
z^2
sage: plot(f(z), 0, 2)
```

Nun ist $f(z)$‘ ein symbolischer Ausdruck. Dies ist unser nächster Stichpunkt unserer Aufzählung.

2. Definition eines "aufrufbaren symbolischen Ausdrucks". Diese können geplottet, differenziert und integriert werden.

```
sage: g(x) = x^2
sage: g        # g bildet x auf x^2 ab
x |--> x^2
sage: g(3)
9
sage: Dg = g.derivative(); Dg
x |--> 2*x
sage: Dg(3)
6
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
sage: plot(g, 0, 2)
```

Beachten Sie, dass während `g` ein aufrufbarer symbolischer Ausdruck ist, `g(x)` ein verwandtes aber unterschiedliches Objekt ist, welches auch geplottet, differenziert, usw. werden kann - wenn auch mit einigen Problemen: Lesen Sie den 5. Stichpunkt unterhalb, um eine Erläuterung zu erhalten.

```
sage: g(x)
x^2
sage: type(g(x))
<type 'sage.symbolic.expression.Expression'>
sage: g(x).derivative()
2*x
sage: plot(g(x), 0, 2)
```

3. Benutzung einer vordefinierten ‘trigonometrischen Sage-Funktion’. Diese können mit ein wenig Hilfestellung differenziert und integriert werden.

```
sage: type(sin)
<class 'sage.functions.trig.Function_sin'>
sage: plot(sin, 0, 2)
sage: type(sin(x))
<type 'sage.symbolic.expression.Expression'>
sage: plot(sin(x), 0, 2)
```

Alleinestehend kann `sin` nicht differenziert werden, zumindest nicht um `cos` zu erhalten.

```
sage: f = sin
sage: f.derivative()
Traceback (most recent call last):
...
AttributeError: ...
```

`f = sin(x)` anstelle von `sin` zu benutzen funktioniert, aber es ist wohl noch besser `f(x) = sin(x)` zu benutzen, um einen aufrufbaren symbolischen Ausdruck zu definieren.

```
sage: S(x) = sin(x)
sage: S.derivative()
x |--> cos(x)
```

Hier sind ein paar häufige Probleme mit Erklärungen:

4. Versehentliche Auswertung.

```
sage: def h(x):
...       if x<2:
...           return 0
...       else:
...           return x-2
```

Das Problem: `plot(h(x), 0, 4)` zeichnet die Linie $y = x - 2$ und nicht die mehrzeilige Funktion, welche durch `h` definiert wird. Der Grund? In dem Befehl `plot(h(x), 0, 4)` wird zuerst `h(x)` ausgewertet: Das bedeutet, dass `x` in die Funktion `h` eingesetzt wird, was wiederum bedeutet, dass `x<2` ausgewertet wird.

```
sage: type(x<2)
<type 'sage.symbolic.expression.Expression'>
```

Wenn eine symbolische Gleichung ausgewertet wird, wie in der Definition von `h`, wird falls sie nicht offensichtlicherweise wahr ist, False zurück gegeben. Also wird `h(x)` zu `x-2` ausgewertet und dies ist die Funktion, die geplottet wird.

Die Lösung: verwenden Sie nicht `plot(h(x), 0, 4)`; benutzen Sie stattdessen:

```
sage: plot(h, 0, 4)
```

5. Versehentliches Erzeugen einer Konstanten anstelle von einer Funktion.

```
sage: f = x
sage: g = f.derivative()
sage: g
1
```

Das Problem: `g(3)`, zum Beispiel, gibt folgenden Fehler zurück: "ValueError: the number of arguments must be less than or equal to 0."

```
sage: type(f)
<type 'sage.symbolic.expression.Expression'>
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
```

`g` ist keine Funktion, es ist eine Konstante, hat also keine zugehörigen Variablen, und man kann in sie nichts einsetzen.

Die Lösung: Es gibt mehrere Möglichkeiten.

- Definieren Sie `f` anfangs als symbolischen Ausdruck.

```
sage: f(x) = x        # statt 'f = x'
sage: g = f.derivative()
sage: g
x |--> 1
sage: g(3)
1
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
```

- Oder mit der ursprünglichen Definition von `f`, definieren Sie `g` als symbolischen Ausdruck.

```
sage: f = x
sage: g(x) = f.derivative()  # statt 'g = f.derivative()'
sage: g
x |--> 1
sage: g(3)
1
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
```

- Oder mit den ursprünglichen Definitionen von `f` and `g`, geben Sie die Variable an, in diese Sie den Wert einsetzen.

```
sage: f = x
sage: g = f.derivative()
sage: g
1
sage: g(x=3)    # statt 'g(3)'
1
```

Schließlich ist hier noch eine Möglichkeit den Unterschied zwischen der Ableitung von `f = x` und der von `f(x) = x` zu erkennen:

```
sage: f(x) = x
sage: g = f.derivative()
sage: g.variables()  # Die in g präsenten Variablen
()
sage: g.arguments()  # Die Argumente die in g gesteckt werden können
(x,)
sage: f = x
sage: h = f.derivative()
```

```
sage: h.variables()
()
sage: h.arguments()
()
```

Wie dieses Beispiel verdeutlichen sollte, nimmt `h` keine Argumente an, und deshalb gibt `h(3)` einen Fehler zurück.

2.7 Wichtige Ringe

Wenn wir Matrizen, Vektoren oder Polynome definieren ist es manchmal nützlich, und manchmal notwendig, den "Ring" anzugeben, über dem diese definiert sind. Ein *Ring* ist ein mathematisches Objekt, für das es die wohldefinierten Operationen Addition und Multiplikation gibt; Falls Sie davon noch nie gehört haben, brauchen Sie wahrscheinlich nur die folgenden vier häufig verwendeten Ringe zu kennen.

- die ganzen Zahlen $\{..., -1, 0, 1, 2, ...\}$, welche `ZZ` in Sage genannt werden.
- die rationalen Zahlen – d.h Brüche oder Quotienten von ganzen Zahlen, welche `QQ` in Sage genannt werden.
- die reellen Zahlen, welche `RR` in Sage genannt werden.
- die komplexen Zahlen, welche `CC` in Sage genannt werden.

Sie müssen diese Unterschiede kennen, da das gleiche Polynom zum Beispiel, unterschiedlich, abhängig von dem Ring über dem es definiert ist, behandelt werden kann. Zum Beispiel hat das Polynom $x^2 - 2$ die beiden Nullstellen $\pm\sqrt{2}$. Diese Nullstellen sind nicht rational, wenn Sie also mit Polynomen über rationalen Koeffizienten arbeiten, lässt sich das Polynom nicht faktorisieren. Mit reellen Koeffizienten lässt es sich faktorisieren. Deshalb müssen Sie den Ring angeben, um sicher zu gehen, dass Sie die Information erhalten, die Sie erwarten. Die folgenden beiden Befehle definieren jeweils die Mengen der Polynome mit rationalen und reellen Koeffizienten. Diese Mengen werden "ratpoly" und "realpoly" genannt, aber das ist hier nicht wichtig; beachten Sie jedoch, dass die Strings ".<t>" und ".<z>" die *Variablen* benennen, die in beiden Fällen benutzt werden.

```
sage: ratpoly.<t> = PolynomialRing(QQ)
sage: realpoly.<z> = PolynomialRing(RR)
```

Jetzt verdeutlichen wir die Behauptung über das Faktorisieren von $x^2 - 2$:

```
sage: factor(t^2-2)
t^2 - 2
sage: factor(z^2-2)
(z - 1.41421356237310) * (z + 1.41421356237310)
```

Ähnliche Kommentare treffen auf Matrizen zu: Die zeilenreduzierte Form eine Matrix kann vom Ring abhängen, über dem sie definiert ist, genauso wie ihre Eigenwerte und Eigenvektoren. Um mehr über das Konstruieren von Polynomen zu erfahren, lesen Sie *Polynome*, und für mehr über Matrizen, lesen Sie *Lineare Algebra*.

Das Symbol `I` steht für die Quadratwurzel von -1; `i` ist ein Synonym für `I`. Natürlich ist dies keine rationale Zahl:

```
sage: i  # Wurzel von -1
I
sage: i in QQ
False
```

Beachten Sie: Der obige Code kann möglicherweise nicht wie erwartet funktionieren. Zum Beispiel wenn der Variablen `i` ein unterschiedlicher Wert, etwa wenn diese als Schleifenvariable verwendet wurde, zugewiesen wurde. Falls dies der Fall ist, tippen Sie

```
sage: reset('i')
```

um den ursprünglichen komplexen Wert der Variable `i` zu erhalten.

Es ist noch eine Feinheit beim Definieren von komplexen Zahlen zu beachten: Wie oben erwähnt wurde, stellt das Symbol `i` eine Quadratwurzel von -1 dar, es ist jedoch eine *formale* oder *symbolische* Quadratwurzel von -1. Das Aufrufen von `CC(i)` oder `CC.0`, gibt die *komplexe* Quadratwurzel von -1 zurück.

```
sage: i = CC(i)        # komplexe Gleitkommazahl
sage: i == CC.0
True
sage: a, b = 4/3, 2/3
sage: z = a + b*i
sage: z
1.33333333333333 + 0.666666666666667*I
sage: z.imag()         # Imaginärteil
0.666666666666667
sage: z.real() == a    # automatische Umwandlung vor dem Vergleich
True
sage: a + b
2
sage: 2*b == a
True
sage: parent(2/3)
Rational Field
sage: parent(4/2)
Rational Field
sage: 2/3 + 0.1        # automatische Umwandlung vor der Addition
0.766666666666667
sage: 0.1 + 2/3        # Umwandlungsregeln sind symmetrisch in SAGE
0.766666666666667
```

Hier sind weitere Beispiele von Ringen in Sage. Wie oben angemerkt, kann auf den Ring der rationalen Zahlen mit `QQ` zugegriffen werden, ebenso wie mit `RationalField()` (ein *Körper* (engl. *field*) ist ein Ring in dem die Multiplikation kommutativ ist, und in dem jedes von Null verschiedene Element in dem Ring einen Kehrwehrt besitzt. Die rationalen Zahlen bilden also auch einen Körper, die ganzen Zahlen jedoch nicht):

```
sage: RationalField()
Rational Field
sage: QQ
Rational Field
sage: 1/2 in QQ
True
```

Die Dezimalzahl `1.2` wird als rationale Zahl in `QQ` gesehen: Dezimalzahlen, die auch rational sind, können in rationale Zahlen "umgewandelt" (engl. "coerced") werden. Die Zahlen π und $\sqrt{2}$ sind jedoch nicht rational:

```
sage: 1.2 in QQ
True
sage: pi in QQ
False
sage: pi in RR
True
sage: sqrt(2) in QQ
False
sage: sqrt(2) in CC
True
```

Für die Verwendung in der höheren Mathematik kennt Sage noch weitere Ringe, wie z.B. endliche Körper, p-adische Zahlen, den Ring der algebraischen Zahlen, Polynomringe und Matrizenringe. Hier sind Konstruktionen einiger von ihnen:

```
sage: GF(3)
Finite Field of size 3
sage: GF(27, 'a')  # Sie müssen den Names des Generators angeben \
....:              # wenn es sich um keinen Primkörper handelt
Finite Field in a of size 3^3
sage: Zp(5)
5-adic Ring with capped relative precision 20
sage: sqrt(3) in QQbar # algebraischer Abschluss von QQ
True
```

2.8 Lineare Algebra

Sage stellt standardmäßige Konstruktionen der Linearen Algebra zur Verfügung. Zum Beispiel das charakteristische Polynom, die Zeilenstufenform, die Spur, die Zerlegung von Matrizen, usw..

Das Erzeugen von Matrizen und die Matrixmultiplikation sind einfach und natürlich:

```
sage: A = Matrix([[1,2,3],[3,2,1],[1,1,1]])
sage: w = vector([1,1,-4])
sage: w*A
(0, 0, 0)
sage: A*w
(-9, 1, -2)
sage: kernel(A)
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1  1 -4]
```

Beachten Sie, dass in Sage der Kern einer Matrix A der "linke Kern", d.h. der Raum der Vektoren w mit $wA = 0$ ist.

Mit der Methode `solve_right` können Matrixgleichungen einfach gelöst werden. Das Auswerten von `A.solve_right(Y)` gibt eine Matrix (oder einen Vektor) X zurück, so dass $AX = Y$ gilt:

```
sage: Y = vector([0, -4, -1])
sage: X = A.solve_right(Y)
sage: X
(-2, 1, 0)
sage: A * X   # wir überprüfen unsere Antwort...
(0, -4, -1)
```

Anstelle von `solve_right` kann auch ein Backslash \ verwendet werden. Benutzen Sie `A \ Y` anstelle von `A.solve_right(Y)`.

```
sage: A \ Y
(-2, 1, 0)
```

Falls keine Lösung existiert, gibt Sage einen Fehler zurück:

```
sage: A.solve_right(w)
Traceback (most recent call last):
...
ValueError: matrix equation has no solutions
```

Auf ähnliche Weisen können Sie `A.solve_left(Y)` benutzen um nach X in $XA = Y$ aufzulösen.

Sage kann auch Eigenwerte und Eigenvektoren berechnen:

```
sage: A = matrix([[0, 4], [-1, 0]])
sage: A.eigenvalues ()
[-2*I, 2*I]
sage: B = matrix([[1, 3], [3, 1]])
sage: B.eigenvectors_left()
[(4, [
(1, 1)
], 1), (-2, [
(1, -1)
], 1)]
```

(Die Syntax der Ausgabe von `eigenvectors_left` ist eine Liste von Tripeln: (Eigenwert, Eigenvektor, Vielfachheit).) Eigenwerte und Eigenvektoren über `QQ` oder `RR` können auch unter Verwendung von Maxima berechnen werden (Lesen Sie *Maxima* unterhalb).

Wie in *Wichtige Ringe* bemerkt wurde, beeinflusst der Ring, über dem die Matrix definiert ist, einige ihrer Eigenschaften. Im Folgenden gibt erste Argument des `matrix`-Befehls Sage zu verstehen, dass die Matrix als Matrix über den ganzen Zahlen (`ZZ`), als Matrix über den rationalen Zahlen (`QQ`), oder als Matrix über den reellen Zahlen (`RR`), aufgefasst werden soll:

```
sage: AZ = matrix(ZZ, [[2,0], [0,1]])
sage: AQ = matrix(QQ, [[2,0], [0,1]])
sage: AR = matrix(RR, [[2,0], [0,1]])
sage: AZ.echelon_form()
[2 0]
[0 1]
sage: AQ.echelon_form()
[1 0]
[0 1]
sage: AR.echelon_form()
[ 1.00000000000000 0.000000000000000]
[0.000000000000000  1.00000000000000]
```

Um Eigenwerte und Eigenvektoren mit reellen oder komplexen Gleitkommazahlen zu berechnen sollte die Matrix über `RDF` (Real Double Field = Körper der reellen Gleitkommazahlen mit doppelter Genauigkeit) oder `CDF` (Complex Double Field = Körper der komplexen Gleitkommazahlen mit doppelter Genauigkeit) definiert werden. Falls kein Koeffizientenring angegeben wird und die Matrixeinträge relle oder komplexe Gleitkommazahlen sind dann werden standardmässig die Körper `RR` oder `CC` verwendet, welche allerdings nicht alle der folgenden Berechnungen unterstützen:

```
sage: ARDF = matrix(RDF, [[1.2, 2], [2, 3]])
sage: ARDF.eigenvalues()
[-0.0931712199461, 4.29317121995]
sage: ACDF = matrix(CDF, [[1.2, I], [2, 3]])
sage: ACDF.eigenvectors_right()
[(0.881845698329 - 0.820914065343*I, [(0.750560818381, -0.616145932705 + 0.238794153033*I)], 1),
(3.31815430167 + 0.820914065343*I, [(0.145594698293 + 0.37566908585*I, 0.915245825866)], 1)]
```

2.8.1 Matrizenräume

Wir erzeugen den Raum $\mathrm{Mat}_{3\times 3}(\mathbf{Q})$ der 3×3 Matrizen mit rationalen Einträgen:

```
sage: M = MatrixSpace(QQ,3)
sage: M
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
```

(Um den Raum der 3 mal 4 Matrizen anzugeben würden Sie `MatrixSpace(QQ,3,4)` benutzen. Falls die

Anzahl der Spalten nicht angegeben wurde, ist diese standardmäßig gleich der Anzahl der Zeilen, so dass `MatrixSpace(QQ,3)` ein Synonym für `MatrixSpace(QQ,3,3)` ist.) Der Matrizenraum hat eine Basis, die Sage als Liste speichert:

```
sage: B = M.basis()
sage: len(B)
9
sage: B[1]
[0 1 0]
[0 0 0]
[0 0 0]
```

Wir erzeugen eine Matrix als ein Element von `M`.

```
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
```

Als nächstes berechnen wir die reduzierte Zeilenstufenform und den Kern.

```
sage: A.echelon_form()
[ 1  0 -1]
[ 0  1  2]
[ 0  0  0]
sage: A.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

Nun zeigen wir, wie man Matrizen berechnen, die über endlichen Körpern definiert sind:

```
sage: M = MatrixSpace(GF(2),4,8)
sage: A = M([1,1,0,0, 1,1,1,1, 0,1,0,0, 1,0,1,1,
...          0,0,1,0, 1,1,0,1, 0,0,1,1, 1,1,1,0])
sage: A
[1 1 0 0 1 1 1 1]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1]
[0 0 1 1 1 1 1 0]
sage: rows = A.rows()
sage: A.columns()
[(1, 0, 0, 0), (1, 1, 0, 0), (0, 0, 1, 1), (0, 0, 0, 1),
 (1, 1, 1, 1), (1, 0, 1, 1), (1, 1, 0, 1), (1, 1, 1, 0)]
sage: rows
[(1, 1, 0, 0, 1, 1, 1, 1), (0, 1, 0, 0, 1, 0, 1, 1),
 (0, 0, 1, 0, 1, 1, 0, 1), (0, 0, 1, 1, 1, 1, 1, 0)]
```

Wir erstellen den Unterraum von $\mathbf{F}_2^8$, der von den obigen Zeilen aufgespannt wird.

```
sage: V = VectorSpace(GF(2),8)
sage: S = V.subspace(rows)
sage: S
Vector space of degree 8 and dimension 4 over Finite Field of size 2
Basis matrix:
[1 0 0 0 0 1 0 0]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1]
[0 0 0 1 0 0 1 1]
sage: A.echelon_form()
```

```
[1 0 0 0 0 1 0 0]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1]
[0 0 0 1 0 0 1 1]
```

Die Basis von S, die von Sage benutzt wird, wird aus den von Null verschiedenen Zeilen der reduzierten Zeilenstufenform der Matrix der Generatoren von S erhalten.

2.8.2 Lineare Algebra mit dünnbesetzten Matrizen

Sage unterstützt Lineare Algebra mit dünnbesetzten Matrizen über Hauptidealringen.

```
sage: M = MatrixSpace(QQ, 100, sparse=True)
sage: A = M.random_element(density = 0.05)
sage: E = A.echelon_form()
```

Der multi-modulare Algorithmus kann bei quadratischen Matrizen gut angewendet werden (bei nicht quadratischen Matrizen ist er nicht so gut):

```
sage: M = MatrixSpace(QQ, 50, 100, sparse=True)
sage: A = M.random_element(density = 0.05)
sage: E = A.echelon_form()
sage: M = MatrixSpace(GF(2), 20, 40, sparse=True)
sage: A = M.random_element()
sage: E = A.echelon_form()
```

Beachten Sie, dass Python zwischen Klein- und Großschreibung unterscheidet:

```
sage: M = MatrixSpace(QQ, 10,10, Sparse=True)
Traceback (most recent call last):
...
TypeError: __classcall__() got an unexpected keyword argument 'Sparse'
```

2.9 Polynome

In diesem Abschnitt erläutern wir, wie man in Sage Polynome erzeugt und benutzt.

2.9.1 Polynome in einer Unbestimmten

Es gibt drei Möglichkeiten Polynomringe zu erzeugen.

```
sage: R = PolynomialRing(QQ, 't')
sage: R
Univariate Polynomial Ring in t over Rational Field
```

Dies erzeugt einen Polynomring und teilt Sage mit (den String) ‘t’ als Unbestimmte bei Ausgaben auf dem Bildschirm zu verwenden. Jedoch definiert dies nicht das Symbol `t` zur Verwendung in Sage, Sie können es also nicht verwenden um ein Polynom (wie z.B. $t^2 + 1$) einzugeben, welches zu `R` gehört.

Eine alternative Möglichkeit ist:

```
sage: S = QQ['t']
sage: S == R
True
```

Dies verhält sich bezüglich `t` gleich.

Eine dritte sehr bequeme Möglichkeit ist:

```
sage: R.<t> = PolynomialRing(QQ)
```

oder

```
sage: R.<t> = QQ['t']
```

oder sogar nur

```
sage: R.<t> = QQ[]
```

Dies hat den zusätzlichen Nebeneffekt, dass die Variable `t` als Unbestimmte des Polynomrings definiert wird, Sie können daher nun wie folgt auf einfache Weise Elemente von `R` definieren. (Beachten Sie, dass die dritte Möglichkeit sehr ähnlich zu der Konstruktor-Notation in Magma ist und, genau wie in Magma, kann Sie dazu verwendet werden eine Vielzahl von Objekten zu definieren.)

```
sage: poly = (t+1) * (t+2); poly
t^2 + 3*t + 2
sage: poly in R
True
```

Unabhängig davon wie Sie den Polynomring definieren, können Sie die Unbestimmte als den 0th Erzeuger zurückerhalten:

```
sage: R = PolynomialRing(QQ, 't')
sage: t = R.0
sage: t in R
True
```

Beachten Sie, dass Sie bei den komplexen Zahlen eine ähnliche Konstruktion verwenden können: Sie können die komplexen Zahlen ansehen, als wären sie von dem Symbol `i` über den reellen Zahlen erzeugt; wir erhalten also Folgendes:

```
sage: CC
Complex Field with 53 bits of precision
sage: CC.0  # 0th generator of CC
1.00000000000000*I
```

Beim Erzeugen von Polynomringen kann man sowohl den Ring, als auch den Erzeuger, oder nur den Erzeuger wie folgt erhalten:

```
sage: R, t = QQ['t'].objgen()
sage: t    = QQ['t'].gen()
sage: R, t = objgen(QQ['t'])
sage: t    = gen(QQ['t'])
```

Schließlich treiben wir etwas Arithmetik in $\mathbf{Q}[t]$.

```
sage: R, t = QQ['t'].objgen()
sage: f = 2*t^7 + 3*t^2 - 15/19
sage: f^2
4*t^14 + 12*t^9 - 60/19*t^7 + 9*t^4 - 90/19*t^2 + 225/361
sage: cyclo = R.cyclotomic_polynomial(7); cyclo
t^6 + t^5 + t^4 + t^3 + t^2 + t + 1
sage: g = 7 * cyclo * t^5 * (t^5 + 10*t + 2)
sage: g
7*t^16 + 7*t^15 + 7*t^14 + 7*t^13 + 77*t^12 + 91*t^11 + 91*t^10 + 84*t^9
       + 84*t^8 + 84*t^7 + 84*t^6 + 14*t^5
```

```
sage: F = factor(g); F
(7) * t^5 * (t^5 + 10*t + 2) * (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1)
sage: F.unit()
7
sage: list(F)
[(t, 5), (t^5 + 10*t + 2, 1), (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1, 1)]
```

Beachten Sie, dass die Faktorisierung die Einheit korrekt in Betracht zieht und ausgibt.

Falls Sie zum Beispiel die `R.cyclotomic_polynomial`-Funktion in einem Forschungsprojekt viel verwenden würden, sollten Sie neben Sage zu zitieren, auch versuchen herauszufinden welche Komponente von Sage verwendet wird um das zyklotomische Polynom zu berechnen, und diese ebenso angeben. In diesen Fall sehen Sie im Quellcode der Funktion, welchen Sie mit `R.cyclotomic_polynomial??` erhalten schnell die Zeile `f = pari.polcyclo(n)`, was bedeutet, dass PARI verwendet wird um das zyklotomische Polynom zu berechnen. Zitieren Sie PARI ebenso in Ihrer Arbeit.

Wenn Sie zwei Polynome teilen, erzeugen Sie ein Element des Quotientenkörpers (den Sage automatisch erzeugt).

```
sage: x = QQ['x'].0
sage: f = x^3 + 1; g = x^2 - 17
sage: h = f/g;  h
(x^3 + 1)/(x^2 - 17)
sage: h.parent()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

Mit Hilfe von Laurentreihen können Sie die Reihenentwicklung im Quotientenkörper von `QQ[x]` berechnen:

```
sage: R.<x> = LaurentSeriesRing(QQ); R
Laurent Series Ring in x over Rational Field
sage: 1/(1-x) + O(x^10)
1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + O(x^10)
```

Wenn wir die Variablen unterschiedlich benennen, erhalten wir einen unterschiedlichen Polynomring.

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<y> = PolynomialRing(QQ)
sage: x == y
False
sage: R == S
False
sage: R(y)
x
sage: R(y^2 - 17)
x^2 - 17
```

Der Ring wird durch die Variable bestimmt. Beachten Sie, dass das Erzeugen eines weiteren Rings mit einer `x` genannten Variablen keinen unterschiedlichen Ring zurück gibt.

```
sage: R = PolynomialRing(QQ, "x")
sage: T = PolynomialRing(QQ, "x")
sage: R == T
True
sage: R is T
True
sage: R.0 == T.0
True
```

Sage unterstützt auch Ringe von Potenz- und Laurentreihen über beliebigen Ringen. Im folgenden Beispiel erzeugen wir ein Element aus $\mathbf{F}_7[[T]]$ und teilen es um ein Element aus $\mathbf{F}_7((T))$ zu erhalten.

```
sage: R.<T> = PowerSeriesRing(GF(7)); R
Power Series Ring in T over Finite Field of size 7
sage: f = T  + 3*T^2 + T^3 + O(T^4)
sage: f^3
T^3 + 2*T^4 + 2*T^5 + O(T^6)
sage: 1/f
T^-1 + 4 + T + O(T^2)
sage: parent(1/f)
Laurent Series Ring in T over Finite Field of size 7
```

Sie können einen Potenzreihenring auch mit der Kurzschreibweise, doppelter eckiger Klammern erzeugen:

```
sage: GF(7)[['T']]
Power Series Ring in T over Finite Field of size 7
```

2.9.2 Polynome in mehreren Unbestimmten

Um mit Polynomringen in mehreren Variablen zu arbeiten, deklarieren wir zunächst den Ring und die Variablen.

```
sage: R = PolynomialRing(GF(5),3,"z") # here, 3 = number of variables
sage: R
Multivariate Polynomial Ring in z0, z1, z2 over Finite Field of size 5
```

Genau wie bei dem Definieren von Polynomringen in einer Variablen, gibt es mehrere Möglichkeiten:

```
sage: GF(5)['z0, z1, z2']
Multivariate Polynomial Ring in z0, z1, z2 over Finite Field of size 5
sage: R.<z0,z1,z2> = GF(5)[]; R
Multivariate Polynomial Ring in z0, z1, z2 over Finite Field of size 5
```

Falls die Variablennamen nur einen Buchstaben lang sein sollen, können Sie auch die folgende Kurzschreibweise verwenden:

```
sage: PolynomialRing(GF(5), 3, 'xyz')
Multivariate Polynomial Ring in x, y, z over Finite Field of size 5
```

Als nächstes treiben wir wieder etwas Arithmetik.

```
sage: z = GF(5)['z0, z1, z2'].gens()
sage: z
(z0, z1, z2)
sage: (z[0]+z[1]+z[2])^2
z0^2 + 2*z0*z1 + z1^2 + 2*z0*z2 + 2*z1*z2 + z2^2
```

Sie können auch eine mathematisch etwas weiter verbreitete Schreibweise verwenden um den Polynomring zu definieren.

```
sage: R = GF(5)['x,y,z']
sage: x,y,z = R.gens()
sage: QQ['x']
Univariate Polynomial Ring in x over Rational Field
sage: QQ['x,y'].gens()
(x, y)
sage: QQ['x'].objgens()
(Univariate Polynomial Ring in x over Rational Field, (x,))
```

Polynomringe in mehreren Variablen sind in Sage mit Hilfe von Python-Dictionaries und der "distributiven Darstellung" eines Polynoms implementiert. Sage benutzt Singular [Si], zum Beispiel bei der Berechnung von ggTs und Gröbnerbasen von Idealen.

```
sage: R, (x, y) = PolynomialRing(RationalField(), 2, 'xy').objgens()
sage: f = (x^3 + 2*y^2*x)^2
sage: g = x^2*y^2
sage: f.gcd(g)
x^2
```

Als nächstes erstellen wir das Ideal (f, g) welches von f und g erzeugt wird, indem wir einfach `(f,g)` mit `R` multiplizieren (wir könnten auch `ideal([f,g])` oder `ideal(f,g)`) schreiben.

```
sage: I = (f, g)*R; I
Ideal (x^6 + 4*x^4*y^2 + 4*x^2*y^4, x^2*y^2) of Multivariate Polynomial
Ring in x, y over Rational Field
sage: B = I.groebner_basis(); B
[x^6, x^2*y^2]
sage: x^2 in I
False
```

Übrigens ist die obige Gröbnerbasis keine Liste, sondern eine unveränderliche Folge. Das bedeutet das sie die Attribute "universe" und "parent" besitzt und nicht verändert werden kann. (Dies ist nützlich, da nach dem Ändern der Basis andere Routinen, welche die Gröbnerbasis verwenden, nicht mehr funktionieren könnten)

```
sage: B.parent()
Category of sequences in Multivariate Polynomial Ring in x, y over Rational
Field
sage: B.universe()
Multivariate Polynomial Ring in x, y over Rational Field
sage: B[1] = x
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Etwas (damit meinen wir: nicht so viel wie wir gerne hätten) kommutative Algebra ist in Sage, mit Hilfe von Singular implementiert, vorhanden. Zum Beispiel können wir die Zerlegung in Primideale und die assoziierten Primideale von I berechnen.

```
sage: I.primary_decomposition()
[Ideal (x^2) of Multivariate Polynomial Ring in x, y over Rational Field,
 Ideal (y^2, x^6) of Multivariate Polynomial Ring in x, y over Rational Field]
sage: I.associated_primes()
[Ideal (x) of Multivariate Polynomial Ring in x, y over Rational Field,
 Ideal (y, x) of Multivariate Polynomial Ring in x, y over Rational Field]
```

2.10 Endliche und abelsche Gruppen

Sage unterstützt das Rechnen mit Permutationsgruppen, endlichen klassischen Gruppen (wie z.B. $SU(n,q)$), endlichen Matrixgruppen (mit Ihren eigenen Erzeugern), und abelschen Gruppen (sogar unendlichen). Vieles davon ist mit Hilfe der GAP-Schnittstelle implementiert.

Zum Beispiel können Sie, um eine Permuationsgruppe zu erzeugen, die Liste der Erzeuger wie folgt angeben.

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(3,4)'])
sage: G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
sage: G.order()
120
sage: G.is_abelian()
False
```

```
sage: G.derived_series()           # random-ish output
[Permutation Group with generators [(1,2,3)(4,5), (3,4)],
 Permutation Group with generators [(1,5)(3,4), (1,5)(2,4), (1,3,5)]]
sage: G.center()
Subgroup of (Permutation Group with generators [(3,4), (1,2,3)(4,5)]) generated by [()]
sage: G.random_element()           # random output
(1,5,3)(2,4)
sage: print latex(G)
\langle (3,4), (1,2,3)(4,5) \rangle
```

Sie können in Sage auch die Tabelle der Charaktere (im LaTeX-Format) erhalten:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3)]])
sage: latex(G.character_table())
\left(\begin{array}{rrrr}
1 & 1 & 1 & 1 \\
1 & 1 & -\zeta_{3} - 1 & \zeta_{3} \\
1 & 1 & \zeta_{3} & -\zeta_{3} - 1 \\
3 & -1 & 0 & 0
\end{array}\right)
```

Sage beinhaltet auch klassische und Matrixgruppen über endlichen Körpern:

```
sage: MS = MatrixSpace(GF(7), 2)
sage: gens = [MS([[1,0],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.conjugacy_class_representatives()
(
[1 0]  [0 1]  [0 1]  [0 1]  [0 1]  [0 1]  [0 1]  [0 3]  [0 3]  [0 1]
[0 1], [6 1], [6 3], [6 2], [6 6], [6 4], [6 5], [2 2], [2 5], [6 0],

[6 0]
[0 6]
)
sage: G = Sp(4,GF(7))
sage: G
Symplectic Group of degree 4 over Finite Field of size 7
sage: G.random_element()             # random output
[5 5 5 1]
[0 2 6 3]
[5 0 1 0]
[4 6 3 4]
sage: G.order()
276595200
```

Sie können auch mit (endlichen oder unendlichen) abelschen Gruppen rechnen.

```
sage: F = AbelianGroup(5, [5,5,7,8,9], names='abcde')
sage: (a, b, c, d, e) = F.gens()
sage: d * b**2 * c**3
b^2*c^3*d
sage: F = AbelianGroup(3,[2]*3); F
Multiplicative Abelian group isomorphic to C2 x C2 x C2
sage: H = AbelianGroup([2,3], names="xy"); H
Multiplicative Abelian group isomorphic to C2 x C3
sage: AbelianGroup(5)
Multiplicative Abelian group isomorphic to Z x Z x Z x Z x Z
sage: AbelianGroup(5).order()
+Infinity
```

2.11 Zahlentheorie

Sage hat für Zahlentheorie eine ausgiebige Funktionsvielfalt. Zum Beispiel können wir Arithmetik in $\mathbf{Z}/N\mathbf{Z}$ wie folgt betreiben:

```
sage: R = IntegerModRing(97)
sage: a = R(2) / R(3)
sage: a
33
sage: a.rational_reconstruction()
2/3
sage: b = R(47)
sage: b^20052005
50
sage: b.modulus()
97
sage: b.is_square()
True
```

Sage enthält standardmäßige zahlentheoretische Funktionen. Zum Beispiel:

```
sage: gcd(515,2005)
5
sage: factor(2005)
5 * 401
sage: c = factorial(25); c
15511210043330985984000000
sage: [valuation(c,p) for p in prime_range(2,23)]
[22, 10, 6, 3, 2, 1, 1, 1]
sage: next_prime(2005)
2011
sage: previous_prime(2005)
2003
sage: divisors(28); sum(divisors(28)); 2*28
[1, 2, 4, 7, 14, 28]
56
56
```

Perfekt!

Sages `sigma(n,k)`-Funktion summiert die k^{ten} Potenzen der Teiler von `n`:

```
sage: sigma(28,0); sigma(28,1); sigma(28,2)
6
56
1050
```

Als nächstes illustrieren wir den erweiterten euklidischen Algorithmus, Eulers ϕ-Funktion, und den chinesischen Restsatz:

```
sage: d,u,v = xgcd(12,15)
sage: d == u*12 + v*15
True
sage: n = 2005
sage: inverse_mod(3,n)
1337
sage: 3 * 1337
4011
sage: prime_divisors(n)
```

```
[5, 401]
sage: phi = n*prod([1 - 1/p for p in prime_divisors(n)]); phi
1600
sage: euler_phi(n)
1600
sage: prime_to_m_part(n, 5)
401
```

Als nächstes verifizieren wir ein Beispiel des $3n + 1$ Problems.

```
sage: n = 2005
sage: for i in range(1000):
...       n = 3*odd_part(n) + 1
...       if odd_part(n)==1:
...           print i
...           break
38
```

Schließlich illustrieren wir den chinesischen Restsatz.

```
sage: x = crt(2, 1, 3, 5); x
11
sage: x % 3  # x mod 3 = 2
2
sage: x % 5  # x mod 5 = 1
1
sage: [binomial(13,m) for m in range(14)]
[1, 13, 78, 286, 715, 1287, 1716, 1716, 1287, 715, 286, 78, 13, 1]
sage: [binomial(13,m)%2 for m in range(14)]
[1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
sage: [kronecker(m,13) for m in range(1,13)]
[1, -1, 1, 1, -1, -1, -1, -1, 1, 1, -1, 1]
sage: n = 10000; sum([moebius(m) for m in range(1,n)])
-23
sage: Partitions(4).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

2.11.1 p-adische Zahlen

Der Körper der p-adischen Zahlen ist in Sage implementiert. Beachten Sie, dass sobald Sie einen p-adischer Körper erzeugt haben, dessen Genauigkeit nicht mehr ändern können.

```
sage: K = Qp(11); K
11-adic Field with capped relative precision 20
sage: a = K(211/17); a
4 + 4*11 + 11^2 + 7*11^3 + 9*11^5 + 5*11^6 + 4*11^7 + 8*11^8 + 7*11^9
  + 9*11^10 + 3*11^11 + 10*11^12 + 11^13 + 5*11^14 + 6*11^15 + 2*11^16
  + 3*11^17 + 11^18 + 7*11^19 + O(11^20)
sage: b = K(3211/11^2); b
10*11^-2 + 5*11^-1 + 4 + 2*11 + O(11^18)
```

In die Implementierung von weiteren Ringen von Zahlen über p-adischen Körpern ist viel Arbeit geflossen. Der interessierte Leser ist dazu eingelanden, die Experten in der `sage-support` Google-Gruppe nach weiteren Details zu fragen.

Eine Vielzahl relevanter Methoden sind schon in der NumberField Klasse implementiert.

```
sage: R.<x> = PolynomialRing(QQ)
sage: K = NumberField(x^3 + x^2 - 2*x + 8, 'a')
sage: K.integral_basis()
[1, 1/2*a^2 + 1/2*a, a^2]

sage: K.galois_group(type="pari")
Galois group PARI group [6, -1, 2, "S3"] of degree 3 of the Number Field
in a with defining polynomial x^3 + x^2 - 2*x + 8

sage: K.polynomial_quotient_ring()
Univariate Quotient Polynomial Ring in a over Rational Field with modulus
x^3 + x^2 - 2*x + 8
sage: K.units()
[3*a^2 + 13*a + 13]
sage: K.discriminant()
-503
sage: K.class_group()
Class group of order 1 of Number Field in a with defining polynomial x^3 + x^2 - 2*x + 8
sage: K.class_number()
1
```

2.12 Etwas weiter fortgeschrittene Mathematik

2.12.1 Algebraische Geometrie

Sie können in Sage beliebige algebraische Varietäten definieren, aber manchmal ist die nichttriviale Funktionalität auf Ringe über $\mathbf{Q}$ oder endliche Körper beschränkt. Zum Beispiel können wir die Vereinigung zweier affiner, planarer Kurven berechnen, und dann die Kurven als irreduzible Komponenten der Vereinigung zurück erhalten.

```
sage: x, y = AffineSpace(2, QQ, 'xy').gens()
sage: C2 = Curve(x^2 + y^2 - 1)
sage: C3 = Curve(x^3 + y^3 - 1)
sage: D = C2 + C3
sage: D
Affine Curve over Rational Field defined by
   x^5 + x^3*y^2 + x^2*y^3 + y^5 - x^3 - y^3 - x^2 - y^2 + 1
sage: D.irreducible_components()
[
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^2 + y^2 - 1,
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^3 + y^3 - 1
]
```

Wir können auch alle Punkte im Schnitt der beiden Kurven finden, indem wir diese schneiden und dann die irreduziblen Komponenten berechnen.

```
sage: V = C2.intersection(C3)
sage: V.irreducible_components()
[
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  y - 1,
  x,
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  y,
  x - 1,
```

```
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x + y + 2,
  2*y^2 + 4*y + 3
]
```

Also sind zum Beispiel $(1, 0)$ und $(0, 1)$ auf beiden Kurven (wie man sofort sieht), genauso wie bestimmte (quadratischen) Punkte, deren y Koordinaten $2y^2 + 4y + 3 = 0$ erfüllen.

Sage kann auch das torische Ideal der gedrehten Kubik im dreidimensionalen projektiven Raum berechnen:

```
sage: R.<a,b,c,d> = PolynomialRing(QQ, 4)
sage: I = ideal(b^2-a*c, c^2-b*d, a*d-b*c)
sage: F = I.groebner_fan(); F
Groebner fan of the ideal:
Ideal (b^2 - a*c, c^2 - b*d, -b*c + a*d) of Multivariate Polynomial Ring
in a, b, c, d over Rational Field
sage: F.reduced_groebner_bases ()
[[-c^2 + b*d, -b*c + a*d, -b^2 + a*c],
 [-c^2 + b*d, b^2 - a*c, -b*c + a*d],
 [-c^2 + b*d, b*c - a*d, b^2 - a*c, -c^3 + a*d^2],
 [c^3 - a*d^2, -c^2 + b*d, b*c - a*d, b^2 - a*c],
 [c^2 - b*d, -b*c + a*d, -b^2 + a*c],
 [c^2 - b*d, b*c - a*d, -b^2 + a*c, -b^3 + a^2*d],
 [c^2 - b*d, b*c - a*d, b^3 - a^2*d, -b^2 + a*c],
 [c^2 - b*d, b*c - a*d, b^2 - a*c]]
sage: F.polyhedralfan()
Polyhedral fan in 4 dimensions of dimension 4
```

2.12.2 Elliptische Kurven

Die Funktionalität elliptischer Kurven beinhaltet die meisten von PARIs Funktionen zu elliptischen Kurven, den Zugriff auf die Daten von Cremonas Online-Tabellen (dies benötigt ein optionales Datenbankpaket), die Funktionen von mwrank, d.h. 2-Abstiege mit der Berechnung der vollen Mordell-Weil-Gruppe, der SEA Algorithmus, Berechnung aller Isogenien, viel neuem Code für Kurven über $\mathbf{Q}$ und Teile von Denis Simons "algebraic descent" Software.

Der Befehl `EllipticCurve` zum Erzeugen von Elliptischen Kurven hat viele Formen:

- EllipticCurve([a_1, a_2, a_3, a_4, a_6]): Gibt die elliptische Kurve

  $$y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6,$$

  zurück, wobei die a_i's umgewandelt werden zum Typ von a_1. Falls alle a_i den Typ $\mathbf{Z}$ haben, werden sie zu $\mathbf{Q}$ umgewandelt.
- EllipticCurve([a_4, a_6]): Das Gleiche wie oben, jedoch ist $a_1 = a_2 = a_3 = 0$.
- EllipticCurve(label): Gibt die elliptische Kurve aus der Datenbank von Cremona mit dem angegebenen (neuen!) Cremona-Label zurück. Das Label ist ein String, wie z.B. `"11a"` oder `"37b2"`. Die Buchstaben müssen klein geschrieben sein (um sie von dem alten Label unterscheiden zu können).
- EllipticCurve(j): Gibt die elliptische Kurve mit j-Invariante j zurück.
- EllipticCurve(R, [a_1, a_2, a_3, a_4, a_6]): Erzeugt die elliptische Kurve über dem Ring R mit vorgegebenen a_i's wie oben.

Wir veranschaulichen jede dieser Konstruktionen:

```
sage: EllipticCurve([0,0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field

sage: EllipticCurve([GF(5)(0),0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 + 4*x over Finite Field of size 5

sage: EllipticCurve([1,2])
Elliptic Curve defined by y^2  = x^3 + x + 2 over Rational Field

sage: EllipticCurve('37a')
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field

sage: EllipticCurve_from_j(1)
Elliptic Curve defined by y^2 + x*y = x^3 + 36*x + 3455 over Rational Field

sage: EllipticCurve(GF(5), [0,0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 + 4*x over Finite Field of size 5
```

Das Paar $(0, 0)$ ist ein Punkt auf der elliptischen Kurve E definiert durch $y^2 + y = x^3 - x$. Um diesen Punkt in Sage zu erzeugen, geben Sie `E([0,0])` ein. Sage kann auf einer solchen elliptischen Kurve Punkte addieren (erinnern Sie sich: elliptische Kurven haben eine additive Gruppenstruktur, wobei der unendlich ferne Punkt das Nullelement ist, und drei kollineare Punkte auf der Kurve sich zu Null addieren):

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: P = E([0,0])
sage: P + P
(1 : 0 : 1)
sage: 10*P
(161/16 : -2065/64 : 1)
sage: 20*P
(683916417/264517696 : -18784454671297/4302115807744 : 1)
sage: E.conductor()
37
```

Die elliptischen Kurven über den komplexen Zahlen sind durch die j-Invariante parametrisiert. Sage berechnet j-Invarianten wie folgt:

```
sage: E = EllipticCurve([0,0,0,-4,2]); E
Elliptic Curve defined by y^2 = x^3 - 4*x + 2 over Rational Field
sage: E.conductor()
2368
sage: E.j_invariant()
110592/37
```

Wenn wir eine Kurve mit der gleichen j-Invarianten wie E erstellen, muss diese nicht isomorph zu E sein. Im folgenden Beispiel sind die Kurven nicht isomorph, da ihre Führer unterschiedlich sind.

```
sage: F = EllipticCurve_from_j(110592/37)
sage: F.conductor()
37
```

Jedoch ergibt der Twist von F mit 2 eine isomorphe Kurve.

```
sage: G = F.quadratic_twist(2); G
Elliptic Curve defined by y^2 = x^3 - 4*x + 2 over Rational Field
sage: G.conductor()
2368
```

```
sage: G.j_invariant()
110592/37
```

Wir können die Koeffizienten a_n der zur elliptischen Kurve gehörenden L-Reihe oder der Modulform $\sum_{n=0}^{\infty} a_n q^n$ berechnen. Die Berechnung benutzt die PARI C-Bibliothek:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: print E.anlist(30)
[0, 1, -2, -3, 2, -2, 6, -1, 0, 6, 4, -5, -6, -2, 2, 6, -4, 0, -12, 0, -4,
 3, 10, 2, 0, -1, 4, -9, -2, 6, -12]
sage: v = E.anlist(10000)
```

Alle Koeffizienten a_n bis zu $n \leq 10^5$ zu berechnen dauert nur eine Sekunde:

```
sage: %time v = E.anlist(100000)
CPU times: user 0.98 s, sys: 0.06 s, total: 1.04 s
Wall time: 1.06
```

Elliptische Kurven können mit Hilfe ihres Cremona-Labels konstruiert werden. Dies lädt die Kurve zusammen mit Informationen über ihren Rank, mit Tamagawa Zahlen, Regulatoren, usw..

```
sage: E = EllipticCurve("37b2")
sage: E
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 1873*x - 31833 over Rational
Field
sage: E = EllipticCurve("389a")
sage: E
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x  over Rational Field
sage: E.rank()
2
sage: E = EllipticCurve("5077a")
sage: E.rank()
3
```

Wir können auch direkt auf die Cremona-Datenbank zugreifen.

```
sage: db = sage.databases.cremona.CremonaDatabase()
sage: db.curves(37)
{'a1': [[0, 0, 1, -1, 0], 1, 1], 'b1': [[0, 1, 1, -23, -50], 0, 3]}
sage: db.allcurves(37)
{'a1': [[0, 0, 1, -1, 0], 1, 1],
 'b1': [[0, 1, 1, -23, -50], 0, 3],
 'b2': [[0, 1, 1, -1873, -31833], 0, 1],
 'b3': [[0, 1, 1, -3, 1], 0, 3]}
```

Die Objekte, die aus der Datenbank zurückgegeben werden, sind nicht vom Typ `EllipticCurve`. Sie sind Elemente einer Datenbank und haben ein paar Komponenten, und das war's. Es gibt eine kleine Version von Cremonas Datenbank, die standardmäßig zu Sage gehört und beschränkte Information zu elliptischen Kurven mit Führer ≤ 10000 enthält. Es gibt auch eine große optionale Version, welche ausgiebige Daten zu allen elliptischen Kurven mit Führer bis zu 120000 enthält (Stand Oktober 2005). Es gibt auch ein riesiges (2GB großes) optionales Datenbank-Paket für Sage, das in der Stein-Watkins Datenbank hunderte Millionen von elliptischen Kurven enthält.

2.12.3 Dirichlet-Charaktere

Ein *Dirichlet Charakter* ist die Erweiterung eines Homomorphismus $(\mathbf{Z}/N\mathbf{Z})^* \to R^*$, für einen Ring R, zu der Abbildung $\mathbf{Z} \to R$, welche erhalten wird, wenn man diese ganzen Zahlen x mit $\gcd(N, x) > 1$ nach 0 schickt.

```
sage: G = DirichletGroup(12)
sage: G.list()
[Dirichlet character modulo 12 of conductor 1 mapping 7 |--> 1, 5 |--> 1,
Dirichlet character modulo 12 of conductor 4 mapping 7 |--> -1, 5 |--> 1,
Dirichlet character modulo 12 of conductor 3 mapping 7 |--> 1, 5 |--> -1,
Dirichlet character modulo 12 of conductor 12 mapping 7 |--> -1, 5 |--> -1]
sage: G.gens()
(Dirichlet character modulo 12 of conductor 4 mapping 7 |--> -1, 5 |--> 1,
Dirichlet character modulo 12 of conductor 3 mapping 7 |--> 1, 5 |--> -1)
sage: len(G)
4
```

Nachdem wir dies Gruppe erzeugt haben, erstellen wir als nächstes ein Element und rechnen damit.

```
sage: G = DirichletGroup(21)
sage: chi = G.1; chi
Dirichlet character modulo 21 of conductor 7 mapping 8 |--> 1, 10 |--> zeta6
sage: chi.values()
[0, 1, zeta6 - 1, 0, -zeta6, -zeta6 + 1, 0, 0, 1, 0, zeta6, -zeta6, 0, -1,
 0, 0, zeta6 - 1, zeta6, 0, -zeta6 + 1, -1]
sage: chi.conductor()
7
sage: chi.modulus()
21
sage: chi.order()
6
sage: chi(19)
-zeta6 + 1
sage: chi(40)
-zeta6 + 1
```

Es ist auch möglich die Operation der Galoisgruppe $\text{Gal}(\mathbf{Q}(\zeta_N)/\mathbf{Q})$ auf diesen Charakteren zu berechnen, sowie die Zerlegung in direkte Produkte, die der Faktorisierung des Moduls entsprechen.

```
sage: chi.galois_orbit()
[Dirichlet character modulo 21 of conductor 7 mapping 8 |--> 1, 10 |--> zeta6,
Dirichlet character modulo 21 of conductor 7 mapping 8 |--> 1, 10 |--> -zeta6 + 1]

sage: go = G.galois_orbits()
sage: [len(orbit) for orbit in go]
[1, 2, 2, 1, 1, 2, 2, 1]

sage: G.decomposition()
[
Group of Dirichlet characters of modulus 3 over Cyclotomic Field of order
6 and degree 2,
Group of Dirichlet characters of modulus 7 over Cyclotomic Field of order
6 and degree 2
]
```

Als nächstes konstruieren wir die Gruppe der Dirichlet-Charaktere mod 20, jedoch mit Werten in $\mathbf{Q}(i)$:

```
sage: K.<i> = NumberField(x^2+1)
sage: G = DirichletGroup(20,K)
sage: G
Group of Dirichlet characters of modulus 20 over Number Field in i with defining polynomial x^2 + 1
```

Nun berechnen wir mehrere Invarianten von `G`:

```
sage: G.gens()
(Dirichlet character modulo 20 of conductor 4 mapping 11 |--> -1, 17 |--> 1,
Dirichlet character modulo 20 of conductor 5 mapping 11 |--> 1, 17 |--> i)

sage: G.unit_gens()
(11, 17)
sage: G.zeta()
i
sage: G.zeta_order()
4
```

In diesem Beispiel erzeugen wir einen Dirichlet-Charakter mit Werten in einem Zahlenfeld. Wir geben die Wahl der Einheitswurzel im dritten Argument von `DirichletGroup` an.

```
sage: x = polygen(QQ, 'x')
sage: K = NumberField(x^4 + 1, 'a'); a = K.0
sage: b = K.gen(); a == b
True
sage: K
Number Field in a with defining polynomial x^4 + 1
sage: G = DirichletGroup(5, K, a); G
Group of Dirichlet characters of modulus 5 over Number Field in a with
defining polynomial x^4 + 1
sage: chi = G.0; chi
Dirichlet character modulo 5 of conductor 5 mapping 2 |--> a^2
sage: [(chi^i)(2) for i in range(4)]
[1, a^2, -1, -a^2]
```

Hier teilt `NumberField(x^4 + 1, 'a')` Sage mit, dass es das Symbol "a" beim Ausgeben dessen was `K` ist (ein Zahlenfeld mit definierendem Polynom $x^4 + 1$) benutzen soll. Der Name "a" ist zu diesem Zeitpunkt nicht deklariert. Sobald `a = K.0` (oder äquivalent `a = K.gen()`) evaluiert wurde, repräsentiert das Symbol "a" eine Wurzel des erzeugenden Polynoms $x^4 + 1$.

2.12.4 Modulformen

Sage kann einige Berechnungen im Zusammenhang mit Modulformen durchführen, einschließlich Dimensionsberechnungen, das Berechnen von Räumen von Symbolen von Modulformen, Hecke-Operatoren, und Dekompositionen.

Es stehen mehrere Funktionen für das Berechnen von Dimensionen von Räumen von Modulformen zur Verfügung. Zum Beispiel,

```
sage: dimension_cusp_forms(Gamma0(11),2)
1
sage: dimension_cusp_forms(Gamma0(1),12)
1
sage: dimension_cusp_forms(Gamma1(389),2)
6112
```

Als nächstes illustrieren wir die Berechnung von Hecke-Operatoren auf einem Raum von Modulformen von Level 1 und Gewicht 12.

```
sage: M = ModularSymbols(1,12)
sage: M.basis()
([X^8*Y^2,(0,0)], [X^9*Y,(0,0)], [X^10,(0,0)])
sage: t2 = M.T(2)
sage: t2
Hecke operator T_2 on Modular Symbols space of dimension 3 for Gamma_0(1)
of weight 12 with sign 0 over Rational Field
```

```
sage: t2.matrix()
[ -24    0    0]
[   0  -24    0]
[4860    0 2049]
sage: f = t2.charpoly('x'); f
x^3 - 2001*x^2 - 97776*x - 1180224
sage: factor(f)
(x - 2049) * (x + 24)^2
sage: M.T(11).charpoly('x').factor()
(x - 285311670612) * (x - 534612)^2
```

Wir können auch Räume für $\Gamma_0(N)$ und $\Gamma_1(N)$ erzeugen.

```
sage: ModularSymbols(11,2)
Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with sign
 0 over Rational Field
sage: ModularSymbols(Gamma1(11),2)
Modular Symbols space of dimension 11 for Gamma_1(11) of weight 2 with
sign 0 and over Rational Field
```

Nun berechnen wir ein paar charakteristische Polynome und q-Entwicklungen.

```
sage: M = ModularSymbols(Gamma1(11),2)
sage: M.T(2).charpoly('x')
x^11 - 8*x^10 + 20*x^9 + 10*x^8 - 145*x^7 + 229*x^6 + 58*x^5 - 360*x^4
     + 70*x^3 - 515*x^2 + 1804*x - 1452
sage: M.T(2).charpoly('x').factor()
(x - 3) * (x + 2)^2 * (x^4 - 7*x^3 + 19*x^2 - 23*x + 11)
        * (x^4 - 2*x^3 + 4*x^2 + 2*x + 11)
sage: S = M.cuspidal_submodule()
sage: S.T(2).matrix()
[-2  0]
[ 0 -2]
sage: S.q_expansion_basis(10)
[
    q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 - 2*q^9 + O(q^10)
]
```

Wir können sogar Räume von Modulsymbolen mit Charakteren berechnen.

```
sage: G = DirichletGroup(13)
sage: e = G.0^2
sage: M = ModularSymbols(e,2); M
Modular Symbols space of dimension 4 and level 13, weight 2, character
[zeta6], sign 0, over Cyclotomic Field of order 6 and degree 2
sage: M.T(2).charpoly('x').factor()
(x - zeta6 - 2) * (x - 2*zeta6 - 1) * (x + zeta6 + 1)^2
sage: S = M.cuspidal_submodule(); S
Modular Symbols subspace of dimension 2 of Modular Symbols space of
dimension 4 and level 13, weight 2, character [zeta6], sign 0, over
Cyclotomic Field of order 6 and degree 2
sage: S.T(2).charpoly('x').factor()
(x + zeta6 + 1)^2
sage: S.q_expansion_basis(10)
[
q + (-zeta6 - 1)*q^2 + (2*zeta6 - 2)*q^3 + zeta6*q^4 + (-2*zeta6 + 1)*q^5
  + (-2*zeta6 + 4)*q^6 + (2*zeta6 - 1)*q^8 - zeta6*q^9 + O(q^10)
]
```

Hier ist ein weiteres Beispiel davon wie Sage mit den Operationen von Hecke-Operatoren auf dem Raum von Modul-

formen rechnen kann.

```
sage: T = ModularForms(Gamma0(11),2)
sage: T
Modular Forms space of dimension 2 for Congruence Subgroup Gamma0(11) of
weight 2 over Rational Field
sage: T.degree()
2
sage: T.level()
11
sage: T.group()
Congruence Subgroup Gamma0(11)
sage: T.dimension()
2
sage: T.cuspidal_subspace()
Cuspidal subspace of dimension 1 of Modular Forms space of dimension 2 for
Congruence Subgroup Gamma0(11) of weight 2 over Rational Field
sage: T.eisenstein_subspace()
Eisenstein subspace of dimension 1 of Modular Forms space of dimension 2
for Congruence Subgroup Gamma0(11) of weight 2 over Rational Field
sage: M = ModularSymbols(11); M
Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with sign
0 over Rational Field
sage: M.weight()
2
sage: M.basis()
((1,0), (1,8), (1,9))
sage: M.sign()
0
```

Sei T_p die Bezeichnung der gewöhnlichen Hecke-Operatoren (p prim). Wie operieren die Hecke-Operatoren T_2, T_3, T_5 auf dem Raum der Modulsymbole?

```
sage: M.T(2).matrix()
[ 3  0 -1]
[ 0 -2  0]
[ 0  0 -2]
sage: M.T(3).matrix()
[ 4  0 -1]
[ 0 -1  0]
[ 0  0 -1]
sage: M.T(5).matrix()
[ 6  0 -1]
[ 0  1  0]
[ 0  0  1]
```

Die interaktive Kommandozeile

In den meisten Teilen dieses Tutorials gehen wir davon aus, dass Sie Sage mit dem `sage`-Befehl starten. Dieser startet eine angepasste Version der IPython Kommandozeile und lädt Funktionen und Klassen, sodass sie in der Kommandozeile genutzt werden können. Weitere Anpassungen können Sie in der Datei `$SAGE_ROOT/ipythonrc` vornehmen. Nach dem Start von Sage sehen Sie etwa folgendes:

```
----------------------------------------------------------------------
| SAGE Version 4.5.2, Release Date: 2010-08-05                       |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------


sage:
```

Um Sage zu beenden drücken Sie Strg-D oder geben Sie `quit` oder `exit` ein.

```
sage: quit
Exiting SAGE (CPU time 0m0.00s, Wall time 0m0.89s)
```

Unter "wall time" finden Sie die vergangene Echtzeit (der Uhr an Ihrer Wand). Diese ist nötig, da die CPU Zeit Unterprozesse wie GAP oder Singular nicht berücksichtigt.

(Vermeiden Sie es den Sage Prozess mit `kill -9` in der Konsole zu beenden, da so möglicherweise Unterprozesse wie z.B. Maple-Prozesse nicht beendet oder temporäre Dateien in `$HOME/.sage/tmp` nicht gelöscht würden.)

3.1 Ihre Sage Sitzung

Unter einer Sitzung verstehen wir die Ein- und Ausgaben von Sage vom Starten bis zum Beenden. Sage speichert alle Eingaben mittels IPython. Wenn Sie die interaktive Kommandozeile nutzen (im Gegensatz zur Browser-Oberfläche "Notebook"), so können Sie jederzeit mittels `%hist` eine Liste aller bisher getätigten Eingaben sehen. Sie können auch `?` eingeben, um mehr über IPython zu erfahren. Z.B. "IPython unterstützt Zeilennummerierung ... sowie Ein- und Ausgabezwischenspeicherung. Alle Eingaben werden gespeichert und können in Variablen abgerufen werden (neben der normalen Pfeiltasten-Navigation). Die folgenden globalen Variablen existieren immer (also bitte überschreiben Sie sie nicht!)":

```
_:  letzte Eingabe (interaktive Kommandozeile und Browser-Oberfläche)
__: vorletzte Eingabe (nur in der Kommandozeile)
_oh: Liste aller Eingaben (nur in der Kommandozeile)
```

Hier ein Beispiel:

```
sage: factor(100)
 _1 = 2^2 * 5^2
sage: kronecker_symbol(3,5)
 _2 = -1
sage: %hist   # funktioniert nur in der Kommandozeile, nicht im Browser.
1: factor(100)
2: kronecker_symbol(3,5)
3: %hist
sage: _oh
 _4 = {1: 2^2 * 5^2, 2: -1}
sage: _i1
 _5 = 'factor(ZZ(100))\n'
sage: eval(_i1)
 _6 = 2^2 * 5^2
sage: %hist
1: factor(100)
2: kronecker_symbol(3,5)
3: %hist
4: _oh
5: _i1
6: eval(_i1)
7: %hist
```

Wir lassen die Zeilennummerierung im restlichen Tutorial sowie in der weiteren Sage-Dokumentation weg. Sie können auch eine Liste von Eingaben einer Sitzung in einem Makro für diese Sitzung speichern.

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: M = ModularSymbols(37)
sage: %hist
1: E = EllipticCurve([1,2,3,4,5])
2: M = ModularSymbols(37)
3: %hist
sage: %macro em 1-2
Macro `em` created. To execute, type its name (without quotes).

sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over
Rational Field
sage: E = 5
sage: M = None
sage: em
Executing Macro...
sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over
Rational Field
```

Während Sie die interaktive Kommandozeile nutzen, können Sie jeden UNIX-Kommandozeilenbefehl in Sage ausführen, indem Sie ihm ein Ausrufezeichen `!` voranstellen. Zum Beispiel gibt

```
sage: !ls
auto  example.sage glossary.tex  t  tmp  tut.log  tut.tex
```

den Inhalt des aktuellen Verzeichnisses aus.

In der `PATH`-Variablen steht das Sage "bin" Verzeichnis vorne. Wenn Sie also `gp`, `gap`, `singular`, `maxima`, usw. eingeben, starten Sie die in Sage enthaltenden Versionen.

```
sage: !gp
Reading GPRC: /etc/gprc ...Done.

                           GP/PARI CALCULATOR Version 2.2.11 (alpha)
                  i686 running linux (ix86/GMP-4.1.4 kernel) 32-bit version
...
sage: !singular
                     SINGULAR                             /  Development
 A Computer Algebra System for Polynomial Computations   /   version 3-1-0
                                                       0<
     by: G.-M. Greuel, G. Pfister, H. Schoenemann        \   Mar 2009
FB Mathematik der Universitaet, D-67653 Kaiserslautern    \
```

3.2 Ein- und Ausgaben loggen

Die Sage Sitzung loggen bzw. speichern ist nicht das Gleiche (siehe *Speichern und Laden kompletter Sitzungen*). Um Eingaben (und optional auch Ausgaben) zu loggen nutzen Sie den Befehl `logstart`. Geben Sie `logstart?` ein um weitere Informationen zu erhalten. Sie können diesen Befehl nutzen um alle Eingaben und Ausgaben zu loggen, und diese sogar wiederholen in einer zukünftigen Sitzung (indem Sie einfach die Log-Datei laden).

```
was@form:~$ sage
----------------------------------------------------------------------
| SAGE Version 4.5.2, Release Date: 2010-08-05                       |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------

sage: logstart setup
Activating auto-logging. Current session state plus future input saved.
Filename       : setup
Mode           : backup
Output logging : False
Timestamping   : False
State          : active
sage: E = EllipticCurve([1,2,3,4,5]).minimal_model()
sage: F = QQ^3
sage: x,y = QQ['x,y'].gens()
sage: G = E.gens()
sage:
Exiting SAGE (CPU time 0m0.61s, Wall time 0m50.39s).
was@form:~$ sage
----------------------------------------------------------------------
| SAGE Version 4.5.2, Release Date: 2010-08-05                       |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------

sage: load "setup"
Loading log file <setup> one line at a time...
Finished replaying log file <setup>
sage: E
Elliptic Curve defined by y^2 + x*y  = x^3 - x^2 + 4*x + 3 over Rational
Field
sage: x*y
x*y
sage: G
[(2 : 3 : 1)]
```

Wenn Sie Sage in der Linux KDE Konsole `konsole` verwenden, können Sie Ihre Sitzung wie folgt speichern: Nachdem Sie Sage in `konsole` gestartet haben, wählen Sie "Einstellungen", dann "Verlauf...", dann "auf unbegrenzt" setzen. Wenn Sie soweit sind Ihre Sitzung zu speichern, wählen Sie "Bearbeiten" und dann "Verlauf speichern unter..." und geben einen Namen ein, um den Text ihrer Sitzung auf dem Computer zu speichern. Nach dem Speichern der Datei können Sie jene in einem Editor wie GNU Emacs öffnen und ausdrucken.

3.3 Einfügen ignoriert Eingabeaufforderungen

Stellen Sie sich vor, Sie lesen eine Sitzung von Sage oder Python Berechnungen und wollen sie in Sage kopieren, aber überall sind noch die störenden `>>>` oder `sage:` Eingabeaufforderungen. Tatsächlich können Sie einfach die gewünschte Stelle mit Eingabeaufforderungen in Sage einfügen. Der Sage Parser wird standardmäßig die führenden `>>>` oder `sage:` Eingabeaufforderungen entfernen bevor er es an Python weitergibt. Zum Beispiel:

```
sage: 2^10
1024
sage: sage: sage: 2^10
1024
sage: >>> 2^10
1024
```

3.4 Befehle zur Zeitmessung

Wenn Sie den `%time` Befehl vor eine Eingabe schreiben wird die Zeit, die der Aufruf benötigt, ausgegeben nachdem er gelaufen ist. Zum Beispiel können wir die Laufzeit einer bestimmten Potenzierung auf verschiedene Arten vergleichen. Die unten genannte Laufzeit wird unter Umständen weit von der Laufzeit auf Ihrem Computer oder sogar zwischen verschiedenen Sage Versionen abweichen. Zuerst natives Python:

```
sage: %time a = int(1938)^int(99484)
CPU times: user 0.66 s, sys: 0.00 s, total: 0.66 s
Wall time: 0.66
```

Das bedeutet insgesamt 0,66 Sekunden wurden benötigt und die vergangene "Wall time", also die vergangene Echtzeit (auf Ihrer Wanduhr), betrug auch 0,66 Sekunden. Wenn auf Ihrem Computer viele andere Programme gleichzeitig laufen kann die "Wall time" wesentlich größer als die CPU Zeit sein.

Als nächstes messen wir die Laufzeit der Potenzierung unter Verwendung des nativen Sage Ganzzahl-Typs, der (in Cython implementiert ist und) die GMP Bibliothek nutzt:

```
sage: %time a = 1938^99484
CPU times: user 0.04 s, sys: 0.00 s, total: 0.04 s
Wall time: 0.04
```

Unter Verwendung der PARI C-Bibliothek:

```
sage: %time a = pari(1938)^pari(99484)
CPU times: user 0.05 s, sys: 0.00 s, total: 0.05 s
Wall time: 0.05
```

GMP ist also ein bisschen besser (wie erwartet, da die für Sage verwendete PARI Version GMP für Ganzzahlarithmetik nutzt). Sie können ebenso Befehlsblöcke messen, indem Sie `cputime` wie unten verwenden:

```
sage: t = cputime()
sage: a = int(1938)^int(99484)
sage: b = 1938^99484
sage: c = pari(1938)^pari(99484)
```

```
sage: cputime(t)                          # random output
0.64

sage: cputime?
...
    Return the time in CPU second since SAGE started, or with optional
    argument t, return the time since time t.
    INPUT:
        t -- (optional) float, time in CPU seconds
    OUTPUT:
        float -- time in CPU seconds
```

Der `walltime` Befehl entspricht `cputime`, nur misst dieser die Echtzeit.

Wir können die oben genannte Potenz auch in einigen der Computer Algebra Systeme, die Sage mitbringt berechnen. In jedem Fall führen wir einen trivialen Befehl aus, um den entsprechenden Server dieses Programms zu starten. Sollte es erhebliche Unterschiede zwischen Echtzeit und CPU-Zeit geben, deutet dies auf ein Leistungsproblem hin, dem man nachgehen sollte.

```
sage: time 1938^99484;
CPU times: user 0.01 s, sys: 0.00 s, total: 0.01 s
Wall time: 0.01
sage: gp(0)
0
sage: time g = gp('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.04
sage: maxima(0)
0
sage: time g = maxima('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.30
sage: kash(0)
0
sage: time g = kash('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.04
sage: mathematica(0)
        0
sage: time g = mathematica('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.03
sage: maple(0)
0
sage: time g = maple('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.11
sage: gap(0)
0
sage: time g = gap.eval('1938^99484;;')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 1.02
```

Achten Sie darauf, dass GAP und Maxima am langsamsten in diesem Test sind (er lief auf dem Computer `sage.math.washington.edu`). Aufgrund des Pexpect-Schnittstellen-Overheads ist es aber vielleicht unfair diese mit Sage zu vergleichen, welches am schnellsten war.

3.5 Fehlerbehandlung

Wenn irgendetwas schief geht, werden Sie normalerweise eine Python-Fehlermeldung sehen. Python macht sogar einen Vorschlag, was den Fehler ausgelöst hat. Oft sehen Sie den Namen der Fehlermeldung, z.B. `NameError` oder `ValueError` (vgl. Python Reference Manual [Py] für eine komplette Liste der Fehlermeldungen). Zum Beispiel:

```
sage: 3_2
------------------------------------------------------------
   File "<console>", line 1
     ZZ(3)_2
           ^
SyntaxError: invalid syntax

sage: EllipticCurve([0,infinity])
------------------------------------------------------------
Traceback (most recent call last):
...
TypeError: Unable to coerce Infinity (<class 'sage...Infinity'>) to Rational
```

Der interaktive Debugger ist manchmal hilfreich um zu verstehen was schiefgelaufen ist. Sie können ihn ein- oder ausschalten indem Sie `%pdb` eingeben (standardmäßig ist er ausgeschaltet). Die Eingabeaufforderung `ipdb>` erscheint wenn eine Fehlermeldung geworfen wird und der Debugger eingeschaltet ist. Im Debugger können Sie den Status jeder lokalen Variable ausgeben oder im Ausführungstack hoch- und runterspringen. Zum Beispiel:

```
sage: %pdb
Automatic pdb calling has been turned ON
sage: EllipticCurve([1,infinity])
---------------------------------------------------------------------------
<type 'exceptions.TypeError'>             Traceback (most recent call last)
...

ipdb>
```

Tippen Sie `?` in der `ipdb>`-Eingabeaufforderung um eine Liste der Befehle des Debuggers zu erhalten.

```
ipdb> ?

Documented commands (type help <topic>):
========================================
EOF    break  commands   debug    h       l     pdef   quit    tbreak
a      bt     condition  disable  help    list  pdoc   r       u
alias  c      cont       down     ignore  n     pinfo  return  unalias
args   cl     continue   enable   j       next  pp     s       up
b      clear  d          exit     jump    p     q      step    w
whatis where

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

Drücken Sie Strg-D oder geben Sie `quit` ein um zu Sage zurückzukehren.

3.6 Rückwärtssuche und Tab-Vervollständigung

Definieren Sie zuerst einen dreidimensionalen Vektorraum $V = \mathbf{Q}^3$ wie folgt:

```
sage: V = VectorSpace(QQ,3)
sage: V
Vector space of dimension 3 over Rational Field
```

Sie können auch die folgende verkürzte Schreibweise verwenden:

```
sage: V = QQ^3
```

Schreiben Sie den Anfang eines Befehls und drücken Sie dann `Strg-p` (oder drücken Sie einfach die Pfeil-nach-oben-Taste) um zur vorher eingegebenen Zeile zu gelangen, die ebenfalls so beginnt. Das funktioniert auch nach einem kompletten Sage-Neustart noch. Sie können den Verlauf auch mit `Strg-r` rückwärts durchsuchen. Diese Funktionalität wird vom `readline`-Paket bereitgestellt, welches in nahezu jeder Linux-Distribution verfügbar ist.

Es ist sehr einfach alle Unterfunktionen für V mittels Tab-Vervollständigung aufzulisten, indem Sie erst `V.` eingeben, und dann die `[Tabulator Taste]` drücken:

```
sage: V.[tab key]
V._VectorSpace_generic__base_field
...
V.ambient_space
V.base_field
V.base_ring
V.basis
V.coordinates
...
V.zero_vector
```

Wenn Sie die ersten paar Buchstaben einer Funktion tippen und dann die `[Tabulator Taste]` drücken, bekommen Sie nur die Funktionen, die so beginnen angezeigt.

```
sage: V.i[tab key]
V.is_ambient  V.is_dense    V.is_full     V.is_sparse
```

Wenn sie wissen wollen, was eine bestimmte Funktion tut, z.B. die "coordinates"-Funktion, so geben Sie `V.coordinates?` ein um die Hilfe, und `V.coordinates??` um den Quelltext der Funktion zu sehen.

3.7 Integriertes Hilfesystem

Sage hat ein integriertes Hilfesystem. Hängen Sie an einen beliebigen Funktionsnamen ein `?` an, um die Dokumentation dazu aufzurufen.

```
sage: V = QQ^3
sage: V.coordinates?
Type:           instancemethod
Base Class:     <type 'instancemethod'>
String Form:    <bound method FreeModule_ambient_field.coordinates of Vector
space of dimension 3 over Rational Field>
Namespace:      Interactive
File:           /home/was/s/local/lib/python2.4/site-packages/sage/modules/f
ree_module.py
Definition:     V.coordinates(self, v)
Docstring:
    Write v in terms of the basis for self.
```

```
    Returns a list c such that if B is the basis for self, then

            sum c_i B_i = v.

    If v is not in self, raises an ArithmeticError exception.

    EXAMPLES:
        sage: M = FreeModule(IntegerRing(), 2); M0,M1=M.gens()
        sage: W = M.submodule([M0 + M1, M0 - 2*M1])
        sage: W.coordinates(2*M0-M1)
        [2, -1]
```

Wie Sie sehen, beinhaltet die Ausgabe den Typ des Objekts, den Dateinamen in welcher die Funktion definiert ist und eine Beschreibung der Funktionalität mit Beispielen, die Sie direkt in Ihre aktuelle Sitzung einfügen können. Fast alle dieser Beispiele werden regelmäßig automatisch getestet um sicherzustellen, dass sie genau wie beschrieben funktionieren.

Eine andere Funktionalität, die sehr eng in Verbindung mit Open-Source-Gedanken steht ist, dass Sie sich zu jeder Funktion den Quelltext anzeigen lassen können. Sei `f` eine Sage oder Python Funktion, dann können Sie mit `f??` den Quellcode, der `f` definiert anzeigen. Zum Beispiel:

```
sage: V = QQ^3
sage: V.coordinates??
Type:           instancemethod
...
Source:
def coordinates(self, v):
        """
        Write $v$ in terms of the basis for self.
        ...
        """
        return self.coordinate_vector(v).list()
```

Das zeigt uns, dass die `coordinates`-Funktion nichts anderes tut, als `coordinates_vector`-Funktion aufruft und das Ergebnis in eine Liste umwandelt. Aber was tut die `coordinates`-Funktion?

```
sage: V = QQ^3
sage: V.coordinate_vector??
...
def coordinate_vector(self, v):
        ...
        return self.ambient_vector_space()(v)
```

Die `coordinate_vector`-Funktion steckt ihre Eingabe in den umgebenden Raum, was zur Folge hat, dass der Koeffizientenvektor von v zur Basis des Vektorraums V ausgerechnet wird. Der Raum V ist schon der umgebende, nämlich gerade $\mathbf{Q}^3$. Es gibt auch eine `coordinate_vector`-Funktion für Unterräume, und sie funktioniert anders. Wir definieren einen Unterraum und schauen uns das an:

```
sage: V = QQ^3; W = V.span_of_basis([V.0, V.1])
sage: W.coordinate_vector??
...
def coordinate_vector(self, v):
        """
         ...
        """
        # First find the coordinates of v wrt echelon basis.
        w = self.echelon_coordinate_vector(v)
        # Next use transformation matrix from echelon basis to
```

```
        # user basis.
        T = self.echelon_to_user_matrix()
        return T.linear_combination_of_rows(w)
```

(Wenn Sie der Meinung sind, dass diese Implementation ineffizient ist, helfen Sie uns bitte unsere Lineare Algebra zu optimieren.)

Sie können auch `help(command_name)` oder `help(class)` eingeben um eine manpage-artige Hilfe zu bekommen.

```
sage: help(VectorSpace)
Help on class VectorSpace ...

class VectorSpace(__builtin__.object)
 |  Create a Vector Space.
 |
 |  To create an ambient space over a field with given dimension
 |  using the calling syntax ...
 :
 :
```

Wenn Sie `q` drücken um das Hilfesystem zu verlassen, kommen Sie genau dahin zurück, wo Sie Ihre Sitzung verlassen haben. Die `help` Anzeige bleibt nicht in Ihrer Sitzung zurück im Gegensatz zu `funktion?`. Es ist besonders hilfreich `help(modul_name)` zu nutzen. Zum Beispiel sind Vektorräume in `sage.modules.free_module` definiert. Geben Sie also `help(sage.modules.free_module)` ein, um die Dokumentation des ganzen Moduls zu sehen. Wenn Sie sich Die Dokumentation mit `help` ansehen, können Sie mit `/` vorwärts und mit `?` rückwärts suchen.

3.8 Speichern und Laden von individuellen Objekten

Angenommen Sie berechnen eine Matrix oder schlimmer, einen komplizierten Modulsymbolraum, und Sie wollen ihn für später speichern. Was können Sie tun? Es gibt mehrere Möglichkeiten für Computer Algebra Systeme solche individuellen Objekte zu speichern.

1. **speichern Ihres Spiels:** Unterstützt nur das Speichern und Laden kompletter Sitzungen (z.B. GAP, Magma).
2. **Einheitliche Ein-/Ausgabe: Bringen Sie jedes Objekt in eine Form, die** Sie wieder einlesen können in (GP/PARI).
3. **Eval**: Machen Sie beliebigen Code auswertbar im Interpreter (z.B. Sigular, PARI).

Da Sage Python nutzt, braucht es einen anderen Ansatz, nämlich dass jedes Objekt serialisiert werden kann. Das heißt es in eine Zeichenkette umzuwandeln, die man wieder einlesen kann. Das ist im Prinzip ähnlich zum einheitlichen Ein-/Ausgabe Ansatz von PARI, abgesehen von der zu komplizierten Darstellung auf dem Bildschirm. Außerdem ist das Laden und Speichern (meistens) vollautomatisch und benötigt nicht einmal speziellen Programmieraufwand; es ist einfach ein Merkmal, das von Grund auf in Python war.

Fast alle Objekte x in Sage können in komprimierter Form gespeichert werden via `save(x, Dateiname)` (oder in vielen Fällen `x.save(Dateiname)`). Um das Objekt wieder zu laden, nutzen Sie `load(Dateiname)`.

```
sage: A = MatrixSpace(QQ,3)(range(9))^2
sage: A
[ 15  18  21]
[ 42  54  66]
[ 69  90 111]
sage: save(A, 'A')
```

Sie sollten Sage nun schließen und neu starten. Dann können Sie `A` wieder laden:

```
sage: A = load('A')
sage: A
[ 15  18  21]
[ 42  54  66]
[ 69  90 111]
```

Sie können das selbe mit komplizierteren Objekten, wie etwa elliptischen Kurven machen. Alle Daten über das Objekt sind zwischengespeichert und werden mit dem Objekt gespeichert. Zum Beispiel:

```
sage: E = EllipticCurve('11a')
sage: v = E.anlist(100000)              # dauert etwas länger
sage: save(E, 'E')
sage: quit
```

Die gespeicherte Version von `E` braucht 153 Kilobyte, da die ersten 100000 a_n mitgespeichert werden.

```
~/tmp$ ls -l E.sobj
-rw-r--r--  1 was was 153500 2006-01-28 19:23 E.sobj
~/tmp$ sage [...]
sage: E = load('E')
sage: v = E.anlist(100000)              # sofort!
```

(In Python wird das Laden und Speichern mittels des `cPickle` Moduls umgesetzt. Genauer: Ein Sage Objekt `x` kann mit `cPickle.dumps(x, 2)` gespeichert werden. Beachten Sie die `2`!)

Sage kann allerdings keine individuellen Objekte anderer Computer Algebra Systeme wie GAP, Singular, Maxima, usw. laden und speichern. Sie sind mit "invalid" gekennzeichnet nach dem Laden. In GAP werden viele Objekte in einer Form dargestellt, die man wiederherstellen kann, viele andere allerdings nicht. Deshalb ist das Wiederherstellen aus ihren Druckdarstellungen nicht erlaubt.

```
sage: a = gap(2)
sage: a.save('a')
sage: load('a')
Traceback (most recent call last):
...
ValueError: The session in which this object was defined is no longer
running.
```

GP/PARI Objekte können hingegen gespeichert und geladen werden, da ihre Druckdarstellung ausreicht um sie wiederherzustellen.

```
sage: a = gp(2)
sage: a.save('a')
sage: load('a')
2
```

Gespeicherte Objekte können auch auf Computern mit anderen Architekturen oder Betriebssystemen wieder geladen werden. Zum Beispiel können Sie eine riesige Matrix auf einem 32 Bit Mac OS X speichern und später auf einem 64 Bit Linux System laden, dort die Stufenform herstellen und dann wieder zurückladen. Außerdem können Sie in den meisten Fällen auch Objekte laden, die mit anderen Sage Versionen gespeichert wurden, solange der Quelltext des Objekts nicht zu verschieden ist. Alle Attribute eines Objekts werden zusammen mit seiner Klasse (aber nicht dem Quellcode) gespeichert. Sollte diese Klasse in einer neueren Sage Version nicht mehr existieren, kann das Objekt in dieser neueren Sage Version nicht mehr geladen werden. Aber Sie könnten es in der alten Sage Version laden, die Objekt Dictionaries mit `x.__dict__` laden und das Objekt zusammen mit diesem in der neuen Sage Version laden.

3.8.1 Als Text speichern

Sie können die ASCII Text Darstellung eines Objekts in eine Klartextdatei schreiben, indem Sie die Datei einfach mit Schreibzugriff öffnen und die Textdarstellung des Objekts hineinkopieren. (Sie können auch viele andere Objekte auf diese Art speichern.) Wenn Sie alle Objekte hineinkopiert haben, schließen Sie die Datei einfach.

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: f = (x+y)^7
sage: o = open('file.txt','w')
sage: o.write(str(f))
sage: o.close()
```

3.9 Speichern und Laden kompletter Sitzungen

Sage hat eine sehr flexible Unterstützung für das Speichern und Laden kompletter Sitzungen.

Der Befehl `save_session(sitzungsname)` speichert alle Variablen, die Sie während dieser Sitzung definiert haben als ein Dictionary `sessionname`. (Im seltenen Fall, dass eine Variable nicht gespeichert werden kann, fehlt sie anschließend einfach im Dictionary.) Die erzeugte Datei ist eine `.sobj`-Datei und kann genau wie jedes andere Objekt geladen werden. Wenn Sie Objekte aus einer Sitzung laden, werden Sie diese in einem Dictionary finden. Dessen Schlüssel sind die Variablen und dessen Werte sind die Objekte.

Sie können den `load_session(sitzungsname)` Befehl nutzen um die Variablen aus `sitzungsname` in die aktuelle Sitzung zu laden. Beachten Sie, dass dieses Vorgehen nicht die Variablen der aktuellen Sitzung löscht, vielmehr werden beide Sitzungen vereinigt.

Starten wir also zunächst Sage und definieren einige Variablen.

```
sage: E = EllipticCurve('11a')
sage: M = ModularSymbols(37)
sage: a = 389
sage: t = M.T(2003).matrix(); t.charpoly().factor()
 _4 = (x - 2004) * (x - 12)^2 * (x + 54)^2
```

Als nächstes speichern wir unsere Sitzung, was jede der Variablen in eine Datei speichert. Dann sehen wir uns die Datei, die etwa 3 Kilobyte groß ist an.

```
sage: save_session('misc')
Saving a
Saving M
Saving t
Saving E
sage: quit
was@form:~/tmp$ ls -l misc.sobj
-rw-r--r--  1 was was 2979 2006-01-28 19:47 misc.sobj
```

Zuletzt starten wir Sage neu, definieren uns eine extra Variable, und laden unsere gespeicherte Sitzung.

```
sage: b = 19
sage: load_session('misc')
Loading a
Loading M
Loading E
Loading t
```

Jede der gespeicherten Variablen ist wieder verfügbar und die Variable `b` wurde nicht überschrieben.

```
sage: M
Full Modular Symbols space for Gamma_0(37) of weight 2 with sign 0
and dimension 5 over Rational Field
sage: E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational
Field
sage: b
19
sage: a
389
```

3.10 Die Notebook Umgebung

Das Sage Browser Notebook wird mit

```
sage: notebook()
```

in der Sage Kommandozeile gestartet. Der Befehl startet das Sage Notebook und ebenso Ihren Standardbrowser. Die Serverstatus-Dateien liegen unter `$HOME/.sage/sage\_notebook`.

Die andere Optionen enthalten z.B.

```
sage: notebook("Verzeichnis")
```

was einen neuen Notebook Server mit den Dateien aus dem angegebenen Verzeichnis startet (anstelle des Standardverzeichnises `$HOME/.sage/sage_notebook`). Das kann hilfreich sein, wenn Sie einige Worksheets für ein Projekt oder verschiedene gleichzeitig laufende Notebook Server von einander trennen wollen.

Wenn Sie das Notebook starten, werden zuerst die folgenden Dateien erzeugt in `$HOME/.sage/sage_notebook`:

```
nb.sobj       (Die notebook SAGE Objekt Datei)
objects/      (Ein Verzeichnis, das SAGE Objekte enthält)
Worksheets/   (Ein Verzeichnis das SAGE Worksheets enthält).
```

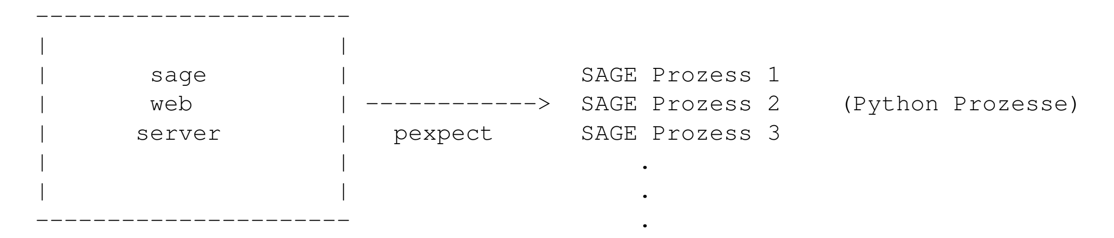
Nach dem Anlegen dieser Dateien, startet das notebook als Webserver.

Ein “Notebook” ist eine Sammlung von Benutzerkonten, von dem jedes verschiedene Worksheets enthalten kann. Wenn Sie ein neues Worksheet erstellen, werden alle zugehörigen Daten unter `Worksheets/username/number` gespeichert. In jedem solchen Verzeichnis ist eine Klartextdatei namens `Worksheet.txt` - sollte mit Ihren Worksheets oder Sage irgendetwas Unvorhergesehenes passieren, enthält diese Datei alles was Sie benötigen um Ihre Worksheets wiederherzustellen.

Innerhalb von Sage können Sie mit `notebook?` mehr Informationen zum Start eines Notebook-Servers erhalten.

Das folgende Diagramm veranschaulicht die Architektur eines Sage Notebooks.

```
----------------------
|                    |
|                    |
|   firefox/safari   |
|                    |
|     javascript     |
|      programm      |
|                    |
|                    |
----------------------
      |       ^
      | AJAX  |
      V       |
```



Um Hilfe zu einem Sage-Befehl `befehl` im Notebook-Browser zu bekommen geben Sie `befehl?` ein und drücken Sie <esc> (nicht <shift-enter>).

Für Informationen zu Tastenbefehlen des Notebook-Browsers klicken Sie auf den `Help` Link.

KAPITEL 4

Schnittstellen

Ein zentraler Aspekt von Sage ist, dass es Berechnungen mit Objekten vieler verschiedener Computer Algebra Systeme unter einem Dach durch eine einheitliche Schnittstelle und Programmiersprache vereinigt.

Die `console` und `interact` Methoden einer Schnittstelle unterstützen viele verschiedene Dinge. Zum Beispiel, anhand von GAP:

1. `gap.console()`: Öffnet die GAP Konsole und übergibt GAP die Kontrolle. Hier ist Sage nichts weiter als ein praktischer Programmstarter, ähnlich einer Linux-Bash-Konsole.
2. `gap.interact()`: Ist eine einfache Art mit einer GAP Instanz zu interagieren, die Sage Objekte enthalten kann. Sie können Sage Objekte in diese GAP Sitzung importieren (sogar von der interaktiven Schnittstelle aus), usw.

4.1 GP/PARI

PARI ist ein kompaktes, sehr ausgereiftes, stark optimiertes C-Programm, dessen primärer Fokus Zahlentheorie ist. Es gibt zwei sehr verschiedene Schnittstellen, die Sie in Sage nutzen können:

- `gp` - Der "**G** o **P** ARI" Interpreter und
- `pari` - Die PARI-C-Bibliothek.

Die folgenden Zeilen zum Beispiel sind zwei Wege, genau das gleiche zu tun. Sie sehen identisch aus, aber die Ausgabe ist verschieden, und was hinter den Kulissen passiert ist völlig unterschiedlich.

```
sage: gp('znprimroot(10007)')
Mod(5, 10007)
sage: pari('znprimroot(10007)')
Mod(5, 10007)
```

Im ersten Fall wird eine separate Kopie des GP-Interpreters als Server gestartet, die Zeichenkette `'znprimroot(10007)'` übergeben, von GP ausgewertet und das Ergebnis wird einer Variable in GP zugewiesen (was Platz im Speicher des GP-Unterprozesses benötigt, der nicht wieder freigegeben wird). Dann wird der Wert der Variablen erst angezeigt. Im zweiten Fall wird kein separates Programm gestartet, stattdessen wird die Zeichenkette `'znprimroot(10007)'` von einer bestimmten PARI-C-Bibliotheksfunktion ausgewertet. Das Ergebnis wird im Speicher von Python gehalten, welcher freigegeben wird wenn die Variable nicht mehr referenziert wird. Die Objekte haben außerdem verschiedene Typen:

```
sage: type(gp('znprimroot(10007)'))
<class 'sage.interfaces.gp.GpElement'>
sage: type(pari('znprimroot(10007)'))
<type 'sage.libs.pari.gen.gen'>
```

Welche Variante sollten Sie also nutzen? Das kommt darauf an was Sie tun. Die GP-Schnittstelle kann alles was ein normales GP/PARI-Konsolenprogramm könnte, da es das Programm startet. Genauergesagt könnten Sie komplizierte PARI-Programme laden und laufen lassen. Im Gegensatz dazu ist die PARI-Schnittstelle (mittels C-Bibliothek) wesentlich restriktiver. Zuerst einmal sind nicht alle Unterfunktionen implementiert. Außerdem wird relativ viel Quellcode nicht in der PARI-Schnittstelle funktionieren, z.B. numerisches Integrieren. Abgesehen davon ist die PARI-Schnittstelle wesentlich schneller und robuster als die von GP.

(Wenn der GP-Schnittstelle der Speicher ausgeht beim Auswerten einer Zeile, wird sie automatisch und unbemerkt den Speicherbereich verdoppeln und das Auswerten erneut versuchen. Dadurch wird Ihre Berechnung nicht abstürzen, falls Sie den benötigen Speicher falsch eingeschätzt haben. Das ist eine hilfreiche Erweiterung, die der gewöhnliche GP-Interpreter nicht bietet. Die PARI-C-Bibliothek hingegen kopiert jedes neue Objekt sofort vom PARI-Stack, daher wird der Stapel nicht größer. Allerdings muss jedes Objekt kleiner als 100 MB sein, da ansonsten der Stapel "überläuft", wenn das Objekt erstellt wird. Dieses zusätzliche Kopieren erzeugt allerdings ein wenig Leistungseinbußen.)

Zusammengefasst nutzt Sage also die PARI-C-Bibliothek um Funktionalitäten eines GP/PARI-Interpreters bereitzustellen, allerdings mit einer anderen komplizierten Speicherverwaltung und der Programmiersprache Python.

Zuerst erstellen wir eine PARI-Liste aus einer Python-Liste.

```
sage: v = pari([1,2,3,4,5])
sage: v
[1, 2, 3, 4, 5]
sage: type(v)
<type 'sage.libs.pari.gen.gen'>
```

Jedes PARI-Objekt ist vom Typ `py_pari.gen`. Den PARI Typ des vorliegenden Objekts können Sie mit der `type` Unterfunktion herausfinden.

```
sage: v.type()
't_VEC'
```

Um eine elliptische Kurve in PARI zu erstellen geben Sie `ellinit([1,2,3,4,5])` ein. Bei Sage ist es ähnlich, nur dass `ellinit` eine Methode ist, die von jedem PARI-Objekt aus aufgerufen werden kann, z.B. unser `t\_VEC v`.

```
sage: e = v.ellinit()
sage: e.type()
't_VEC'
sage: pari(e)[:13]
[1, 2, 3, 4, 5, 9, 11, 29, 35, -183, -3429, -10351, 6128487/10351]
```

Jetzt haben wir eine elliptische Kurve als Objekt und können einige Dinge mit ihr berechnen.

```
sage: e.elltors()
[1, [], []]
sage: e.ellglobalred()
[10351, [1, -1, 0, -1], 1]
sage: f = e.ellchangecurve([1,-1,0,-1])
sage: f[:5]
[1, -1, 0, 4, 3]
```

4.2 GAP

Sage enthält ausserdem GAP für diskrete Mathematik, insbesondere Gruppentheorie.

Hier ist ein Beispiel für GAP's `IdGroup`-Funktion, die die optionale kleine Gruppen Datenbank benötigt, die separat installiert werden muss, wie unten beschrieben.

```
sage: G = gap('Group((1,2,3)(4,5), (3,4))')
sage: G
Group( [ (1,2,3)(4,5), (3,4) ] )
sage: G.Center()
Group( () )
sage: G.IdGroup()    # optional - database_gap
[ 120, 34 ]
sage: G.Order()
120
```

Wir können die gleiche Berechnung in Sage durchführen ohne vorher explizit die GAP-Schnittstelle aufzurufen:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.center()
Subgroup of (Permutation Group with generators [(3,4), (1,2,3)(4,5)]) generated by [()]
sage: G.group_id()    # optional - database_gap
[120, 34]
sage: n = G.order(); n
120
```

(Für einige GAP-Funktionen sollten Sie zwei optionale Sage Pakete installieren. Geben Sie `sage -optional` ein, um eine Liste zu erhalten und wählen Sie das Paket aus, das etwa so aussieht `gap\_packages-x.y.z`. Geben Sie dann `sage -i gap\_packages-x.y.z` ein. Das gleiche machen Sie bitte mit `database\_gap-x.y.z`. Einige nicht-GPL Pakete können installiert werden, indem Sie sie von der GAP-Website [GAPkg] herunter laden und nach `$SAGE_ROOT/local/lib/gap-4.4.10/pkg` entpacken.)

4.3 Singular

Singular bietet eine sehr gute, ausgereifte Bibliothek für Gröbnerbasen, größte gemeinsame Teiler von mehrdimensionalen Polynomen, Basen von Riemann-Roch Räumen einer planaren Kurve und Faktorisierungen unter anderem. Wir zeigen hier die Faktorisierung mehrdimensionaler Polynome mit Sages Singular-Schnittstelle (ohne die `...`):

```
sage: R1 = singular.ring(0, '(x,y)', 'dp')
sage: R1
//   characteristic : 0
//   number of vars : 2
//        block   1 : ordering dp
//                  : names    x y
//        block   2 : ordering C
sage: f = singular('9*y^8 - 9*x^2*y^7 - 18*x^3*y^6 - 18*x^5*y^6 + \
....: 9*x^6*y^4 + 18*x^7*y^5 + 36*x^8*y^4 + 9*x^10*y^4 - 18*x^11*y^2 - \
....: 9*x^12*y^3 - 18*x^13*y^2 + 9*x^16')
```

Wir haben also das Polynom f definiert, nun geben wir es aus und faktorisieren es.

```
sage: f
9*x^16-18*x^13*y^2-9*x^12*y^3+9*x^10*y^4-18*x^11*y^2+36*x^8*y^4+18*x^7*y^5-18*x^5*y^6+9*x^6*y^4-18*x
sage: f.parent()
Singular
sage: F = f.factorize(); F
```

```
[1]:
   _[1]=9
   _[2]=x^6-2*x^3*y^2-x^2*y^3+y^4
   _[3]=-x^5+y^2
[2]:
   1,1,2
sage: F[1][2]
x^6-2*x^3*y^2-x^2*y^3+y^4
```

Genau wie im GAP Beispiel in *GAP*, können wir diese Faktorisierung berechnen ohne explizit die Singular-Schnittstelle zu nutzen. (Dennoch nutzt Sage im Hintergrund die Singular-Schnittstelle für die Berechnung.) Bitte geben Sie ein ohne `...`:

```
sage: x, y = QQ['x, y'].gens()
sage: f = 9*y^8 - 9*x^2*y^7 - 18*x^3*y^6 - 18*x^5*y^6 + 9*x^6*y^4 \
....: + 18*x^7*y^5 + 36*x^8*y^4 + 9*x^10*y^4 - 18*x^11*y^2 - 9*x^12*y^3 \
....: - 18*x^13*y^2 + 9*x^16
sage: factor(f)
(9) * (-x^5 + y^2)^2 * (x^6 - 2*x^3*y^2 - x^2*y^3 + y^4)
```

4.4 Maxima

Das in Lisp implementierte Maxima ist ein Teil von Sage. Hingegen wird das gnuplot-Paket (welches Maxima standardmäßig zum plotten nutzt) als optionales Sage-Paket angeboten. Neben anderen Dingen rechnet Maxima mit Symbolen. Maxima integriert und differenziert Funktionen symbolisch, löst gewöhnliche Differentialgleichungen ersten Grades sowie viele lineare Differentialgleichungen zweiten Grades und besitzt eine Methode zur Laplace Transformation linearer Differentialgleichungen von beliebigem Grad. Maxima kennt eine große Zahl spezieller Funktionen, plottet mittels gnuplot und hat Methoden, um Polynomgleichungen oder Matrizen zu lösen oder zu verändern (z.B. Zeilenelimination oder Eigenwerte und Eigenvektoren berechnen).

Wir zeigen die Sage/Maxima Schnittstelle, indem wir die Matrix konstruieren, deren i, j Eintrag gerade i/j ist, für $i, j = 1, \ldots, 4$.

```
sage: f = maxima.eval('ij_entry[i,j] := i/j')
sage: A = maxima('genmatrix(ij_entry,4,4)'); A
matrix([1,1/2,1/3,1/4],[2,1,2/3,1/2],[3,3/2,1,3/4],[4,2,4/3,1])
sage: A.determinant()
0
sage: A.echelon()
matrix([1,1/2,1/3,1/4],[0,0,0,0],[0,0,0,0],[0,0,0,0])
sage: A.eigenvalues()
[[0,4],[3,1]]
sage: A.eigenvectors()
[[[0,4],[3,1]],[[[1,0,0,-4],[0,1,0,-2],[0,0,1,-4/3]],[[1,2,3,4]]]]
```

Hier ein anderes Beispiel:

```
sage: A = maxima("matrix ([1, 0, 0], [1, -1, 0], [1, 3, -2])")
sage: eigA = A.eigenvectors()
sage: V = VectorSpace(QQ,3)
sage: eigA
[[[-2,-1,1],[1,1,1]],[[[0,0,1]],[[0,1,3]],[[1,1/2,5/6]]]]
sage: v1 = V(sage_eval(repr(eigA[1][0][0]))); lambda1 = eigA[0][0][0]
sage: v2 = V(sage_eval(repr(eigA[1][1][0]))); lambda2 = eigA[0][0][1]
sage: v3 = V(sage_eval(repr(eigA[1][2][0]))); lambda3 = eigA[0][0][2]
```

```
sage: M = MatrixSpace(QQ,3,3)
sage: AA = M([[1,0,0],[1, - 1,0],[1,3, - 2]])
sage: b1 = v1.base_ring()
sage: AA*v1 == b1(lambda1)*v1
True
sage: b2 = v2.base_ring()
sage: AA*v2 == b2(lambda2)*v2
True
sage: b3 = v3.base_ring()
sage: AA*v3 == b3(lambda3)*v3
True
```

Zuletzt noch ein Beispiel wie man Sage zum Plotten mittels `openmath` nutzt. Einige von ihnen wurden (verändert) aus dem Maxima Benutzerhandbuch entnommen.

Ein 2D-Plot verschiedener Funktionen (ohne `...` eingeben):

```
sage: maxima.plot2d('[cos(7*x),cos(23*x)^4,sin(13*x)^3]','[x,0,1]', # not tested
....: '[plot_format,openmath]')
```

Ein "live" 3D-Plot, den man mit der Maus bewegen kann:

```
sage: maxima.plot3d ("2^(-u^2 + v^2)", "[u, -3, 3]", "[v, -2, 2]", # not tested
....: '[plot_format, openmath]')
sage: maxima.plot3d("atan(-x^2 + y^3/4)", "[x, -4, 4]", "[y, -4, 4]", # not tested
....: "[grid, 50, 50]",'[plot_format, openmath]')
```

Der nächste Plot ist das berühmte Möbiusband:

```
sage: maxima.plot3d("[cos(x)*(3 + y*cos(x/2)), sin(x)*(3 + y*cos(x/2)), y*sin(x/2)]", # not tested
....: "[x, -4, 4]", "[y, -4, 4]",
....: '[plot_format, openmath]')
```

Und der letzte ist die berühmte Kleinsche Flasche:

```
sage: maxima("expr_1: 5*cos(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y)+ 3.0) - 10.0")
5*cos(x)*(sin(x/2)*sin(2*y)+cos(x/2)*cos(y)+3.0)-10.0
sage: maxima("expr_2: -5*sin(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y)+ 3.0)")
-5*sin(x)*(sin(x/2)*sin(2*y)+cos(x/2)*cos(y)+3.0)
sage: maxima("expr_3: 5*(-sin(x/2)*cos(y) + cos(x/2)*sin(2*y))")
5*(cos(x/2)*sin(2*y)-sin(x/2)*cos(y))
sage: maxima.plot3d ("[expr_1, expr_2, expr_3]", "[x, -%pi, %pi]", # not tested
....: "[y, -%pi, %pi]", "['grid, 40, 40]",
....: '[plot_format, openmath]')
```

KAPITEL 5

Sage, LaTeX und ihre Freunde

Sage und der TeX-Dialekt LaTeX haben eine sehr synergetische Beziehung. Dieses Kapitel hat das Ziel die Vielfalt an Interaktionen, von den einfachsten bis hin zu den ungewöhnlichen und fast schon magischen, vorzustellen. (Sie sollten also nicht gleich das ganze Kapitel im ersten Durchgang durch das Tutorial lesen.)

5.1 Überblick

Es ist wahrscheinlich am einfachsten die verschiedenen Einsatzmöglichkeiten von LaTeX zu verstehen, wenn man sich die drei grundsätzlichen Methoden in Sage ansieht.

1. Jedes Objekt in Sage muss eine LaTeX Darstellung haben. Sie können diese Darstellung erreichen, indem Sie im Notebook oder der Kommandozeile `latex(foo)` ausführen, wobei `foo` ein Objekt in Sage ist. Die Ausgabe ist eine Zeichenkette, die eine recht genaue Darstellung im mathematischen Modus von TeX bietet (z.B. zwischen jeweils zwei Dollarzeichen). Einige Beispiele hierfür folgen unten.

   So kann Sage effektiv genutzt werden um Teile eines LaTeX-Dokuments zu erstellen: Erstellen oder berechnen Sie ein Objekt in Sage, drucken Sie es mit dem `latex()`-Befehl aus und fügen Sie es in Ihr Dokument ein.

2. Die Notebook Schnittstelle ist konfiguriert MathJax zu nutzen um mathematische Ausdrücke im Browser darzustellen. MathJax ist eine Kollektion aus JavaScript-Routinen und zugehörigen Schriftarten. Es ist also nichts zusätzlich einzustellen um mathematische Ausdrücke in Ihrem Browser anzuzeigen, wenn Sie das Sage-Notebook nutzen.

   MathJax wurde entwickelt um einen großen, aber nicht vollständigen Teil von TeX darstellen zu können. Es gibt keine Unterstützung für Dinge, wie komplizierte Tabellen, Kapiteleinteilung oder Dokument Management, da es für genaues Darstellen von TeX Ausdrücken konzipiert wurde. Die nahtlose Darstellung von mathematischen Ausdrücken im Sage Notebook wird durch Konvertierung der `latex()`-Darstellung in MathJax gewährleistet.

   Da MathJax seine eigenen skalierbaren Schriftarten nutzt, ist es anderen Methoden überlegen, die auf Konvertierung in kleine Bilder beruhen.

3. Sollte in der Sage Kommandozeile oder im Notebook mehr LaTeX-Code vorkommen als MathJax verarbeiten kann, kann eine systemweite Installation von LaTeX aushelfen. Sage beinhaltet fast alles, das Sie brauchen um Sage weiter zu entwickeln und zu nutzen. Eine Ausnahme hierzu ist TeX selbst. In diesen Situationen müssen also TeX und verschiedene Konverter installiert sein, um alle Möglichkeiten nutzen zu können.

Hier führen wir einige grundlegenden Funktionen von `latex()` vor.

```
sage: var('z')
z
sage: latex(z^12)
z^{12}
sage: latex(integrate(z^4, z))
\frac{1}{5} \, z^{5}
sage: latex('a string')
\verb|a|\phantom{\verb!x!}\verb|string|
sage: latex(QQ)
\Bold{Q}
sage: latex(matrix(QQ, 2, 3, [[2,4,6],[-1,-1,-1]]))
\left(\begin{array}{rrr}
2 & 4 & 6 \\
-1 & -1 & -1
\end{array}\right)
```

Grundlegende MathJax Funktionen gibt es im Notebook weitgehend automatisch, aber wir können es teilweise mit Hilfe der `MathJax` Klasse demonstrieren. Die `eval` Funktion dieser Klasse konvertiert ein Sage-Objekt in seine LaTeX-Darstellung und dann in HTML mit der CSS `math` Klasse, die dann MathJax verwendet.

```
sage: from sage.misc.latex import MathJax
sage: mj = MathJax()
sage: var('z')
z
sage: mj(z^12)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}z^{12}</script></html>
sage: mj(QQ)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Q}</script></html
sage: mj(ZZ[x])
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Z}[x]</script></h
sage: mj(integrate(z^4, z))
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}\frac{1}{5} \, z^{5}</s
```

5.2 Grundlegende Nutzung

Wie schon im Überblick angekündigt, ist der einfachste Weg Sage's LaTeX-Unterstützung zu nutzen die `latex()` Funktion um eine legitime LaTeX-Darstellung eines mathematischen Objekts zu erhalten. Diese Zeichenketten können dann in unabhängigen LaTeX-Dokumenten genutzt werden. Das funktioniert im Notebook genauso wie in der Sage-Kommandozeile.

Das andere Extrem ist der `view()`-Befehl. In der Sage-Kommandozeile wird der Befehl `view()` die LaTeX-Darstellung von `foo` in ein einfaches LaTeX Dokument packen, und dann dieses mit der systemweiten TeX-Installation aufrufen. Zuletzt wird das passende Programm zum Anzeigen der Ausgabe von TeX aufgerufen. Welche Version von TeX genutzt wird, und damit auch wie die Ausgabe aussieht und welches Anzeigeprogramm aufgerufen wird, kann angepasst werden (siehe *Anpassen der LaTeX-Verarbeitung*).

Im Notebook schafft der `view(foo)` Befehl die nötige Kombination von HTML und CSS sodass MathJax die LaTeX Darstellung im Arbeitsblatt anzeigt. Für den Anwender erstellt er einfach eine schön formatierte Ausgabe, die sich von der normalen ASCII Ausgabe aus Sage unterscheidet. Nicht jedes mathematische Objekt in Sage hat eine LaTeX-Darstellung, die die eingeschränkten Möglichkeiten von MathJax unterstützt. In diesen Fällen kann die MathJax Darstellung umgangen werden, und stattdessen die systemweite TeX-Installation aufgerufen werden. Dessen Ausgabe kann dann als Bild im Arbeitsblatt angezeigt werden. Die Einstellungen und Auswirkungen dieses Prozesses wird im Kapitel *Anpassen der LaTeX-Generierung* dargestellt.

Der interne `pretty_print()` Befehl zeigt die Konvertierung von Sage Objekten in HTML Code der MathJax nutzt im Notebook.

```
sage: from sage.misc.latex import pretty_print
sage: pretty_print(x^12)
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}x^{12}</script></html>
sage: pretty_print(integrate(sin(x), x))
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}-\cos\left(x\right)</script></html>
```

Das Notebook hat zwei weitere Möglichkeiten TeX zu nutzen. Die erste ist der "Typeset"-Knopf über der ersten Zelle eines Arbeitsblatts, rechts von den vier Drop-Down-Boxen. Ist er ausgewählt werden die Ausgaben aller folgenden Berechnungen von MathJax interpretiert. Beachten Sie, dass dieser Befehl nicht rückwirkend ist – alle vorher berechneten Zellen werden nicht neu berechnet. Im Grunde ist der "Typeset"-Knopf gleichzusetzen mit dem Aufruf des `view()`-Befehls in allen Zellen.

Die zweite Möglichkeit im Notebook ist das Eingeben von TeX Kommentaren in einem Arbeitsblatt. Wenn der Cursor zwischen zwei Zellen steht, und der erscheinende blaue Balken mit gedrückter Shift Taste geklickt wird, wird ein kleiner Texteditor TinyMCE geöffnet. Dieser erlaubt die Eingabe von HTML und CSS formatiertem Text mit einem WYSIWYG-Editor. Es ist also möglich den so formatierten Text als Kommentar in einem Arbeitsblatt unterzubringen. Text den Sie hier zwischen `$...$` oder `$$...$$` eingeben wird ebenfalls von MathJax in einer "inline" bzw. "display math" Umgebung gesetzt.

5.3 Anpassen der LaTeX-Generierung

Es gibt verschiedene Arten den vom `latex()`-Befehl generierten LaTeX-Code anzupassen. Im Notebook und der Sage Kommandozeile gibt es ein vordefiniertes Objekt Namens `latex`, das verschiedene Methoden hat, die Sie sich auflisten lassen können indem Sie `latex.` eingeben und die Tab Taste drücken (beachten Sie den Punkt).

Ein gutes Beispiel ist die `latex.matrix_delimiters` Methode. Es kann benutzt werden um die Darstellung der Matrizen zu beeinflussen – runde Klammern, eckige Klammern, geschwungene Klammern oder senkrechte Striche. Sie müssen sich nicht für eine Darstellung entscheiden, Sie können verschiedene miteinander kombinieren, wie Sie es wünschen. Beachten Sie dass die in LaTeX benötigten Backslashes einen zusätzlichen Slash benötigen damit sie in Python korrekt erkannt werden.

```
sage: A = matrix(ZZ, 2, 2, range(4))
sage: latex(A)
\left(\begin{array}{rr}
0 & 1 \\
2 & 3
\end{array}\right)
sage: latex.matrix_delimiters(left='[', right=']')
sage: latex(A)
\left[\begin{array}{rr}
0 & 1 \\
2 & 3
\end{array}\right]
sage: latex.matrix_delimiters(left='\\{', right='\\}')
sage: latex(A)
\left\{\begin{array}{rr}
0 & 1 \\
2 & 3
\end{array}\right\}
```

Die `latex.vector_delimiters` Methode funktioniert ähnlich.

Die Darstellung von Ringen und Körpern (ganze, rationale, reelle Zahlen, etc.) kann mit der `latex.blackboard_bold` Methode verändert werden. Diese Mengen werden in standardmäßig in fett gedruckt, alternativ können sie auch mit Doppelstrichen geschrieben werden. Hierfür wird das `\Bold{}`-Makro genutzt, das in Sage integriert ist.

```
sage: latex(QQ)
\Bold{Q}
sage: from sage.misc.latex import MathJax
sage: mj=MathJax()
sage: mj(QQ)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Q}</script></html
sage: latex.blackboard_bold(True)
sage: mj(QQ)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbb{#1}}\Bold{Q}</script></html
sage: latex.blackboard_bold(False)
```

Dank der Erweiterbarkeit von TeX können Sie selbst Makros und Pakete einbinden. Individuelle Makros können hinzugefügt werden, die dann von MathJax als TeX-Schnipsel interpretiert werden.

```
sage: latex.extra_macros()
''
sage: latex.add_macro("\\newcommand{\\foo}{bar}")
sage: latex.extra_macros()
'\\newcommand{\\foo}{bar}'
sage: var('x y')
(x, y)
sage: latex(x+y)
x + y
sage: from sage.misc.latex import MathJax
sage: mj=MathJax()
sage: mj(x+y)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}\newcommand{\foo}{bar}x
```

Zusätzliche Makros, die so hinzugefügt wurden, werden auch vom systemweiten TeX genutzt, wenn MathJax an seine Grenzen gestoßen ist. Der Befehl `latex_extra_preamble` kann genutzt werden um eine Präambel eines kompletten LaTeX Dokuments zu erzeugen, das folgende Beispiel zeigt wie. Beachten Sie wiederrum die doppelten Backslashes in den Python Zeichenketten.

```
sage: latex.extra_macros('')
sage: latex.extra_preamble('')
sage: from sage.misc.latex import latex_extra_preamble
sage: print latex_extra_preamble()
\newcommand{\ZZ}{\Bold{Z}}
...
\newcommand{\Bold}[1]{\mathbf{#1}}
sage: latex.add_macro("\\newcommand{\\foo}{bar}")
sage: print latex_extra_preamble()
\newcommand{\ZZ}{\Bold{Z}}
...
\newcommand{\Bold}[1]{\mathbf{#1}}
\newcommand{\foo}{bar}
```

Für größere oder kompliziertere LaTeX-Ausdrücke können mit `latex.add_to_preamble` Pakete (oder ähnliches) zur LaTeX-Präambel hinzugefügt werden. Der zweite Befehl `latex.add_package_to_preamble_if_available` prüft hingegen erst ob das Paket vorhanden ist, bevor es eingebunden wird.

Hier fügen wir das geometry-Paket zur Präambel hinzu, um die Seitenränder einzustellen. Achten Sie wieder auf die doppelten Backslashes in Python.

```
sage: from sage.misc.latex import latex_extra_preamble
sage: latex.extra_macros('')
sage: latex.extra_preamble('')
sage: latex.add_to_preamble('\\usepackage{geometry}')
```

```
sage: latex.add_to_preamble('\\geometry{letterpaper,total={8in,10in}}')
sage: latex.extra_preamble()
'\\usepackage{geometry}\\geometry{letterpaper,total={8in,10in}}'
sage: print latex_extra_preamble()
\usepackage{geometry}\geometry{letterpaper,total={8in,10in}}
\newcommand{\ZZ}{\Bold{Z}}
...
\newcommand{\Bold}[1]{\mathbf{#1}}
```

Ein bestimmtes Paket, dessen Existenz nicht sicher ist, wird wie folgt eingebunden.

```
sage: latex.extra_preamble('')
sage: latex.extra_preamble()
''
sage: latex.add_to_preamble('\\usepackage{foo-bar-unchecked}')
sage: latex.extra_preamble()
'\\usepackage{foo-bar-unchecked}'
sage: latex.add_package_to_preamble_if_available('foo-bar-checked')
sage: latex.extra_preamble()
'\\usepackage{foo-bar-unchecked}'
```

5.4 Anpassen der LaTeX-Verarbeitung

Es ist möglich zu entscheiden welche Variante von TeX für einen systemweiten Aufruf genutzt werden soll, und somit auch wie die Ausgabe aussehen soll. Ebenso ist es möglich zu beeinflussen, ob das Notebook MathJax oder die systemweite LaTeX Installation nutzt.

Der Befehl `latex.engine()` entscheidet, ob die systemweiten Anwendungen `latex`, `pdflatex` oder `xelatex` genutzt werden für kompliziertere LaTeX-Ausdrücke. Wenn `view()` in der Sage Kommandozeile aufgerufen wird, und `latex` als Prozessor eingestellt ist, wird eine .dvi Datei erzeugt, die dann mit einem dvi Anzeigeprogramm (wie xdvi) angezeigt wird. Im Gegensatz hierzu wird bei Aufruf von `view()` mit dem Prozessor `pdflatex` eine .PDF Datei erzeugt, die mit dem Standard-PDF-Programm angezeigt wird. (acrobat, okular, evince, etc.).

Im Notebook kann es nötig sein, dem System die Entscheidung abzunehmen, ob MathJax für einige TeX-Schnipsel, oder das systemweite LaTeX für kompliziertere Ausdrücke genutzt werden soll. Es gibt eine Liste von Befehlen, die wenn einer von ihnen in einem Stück LaTeX-Code erkannt wird, die Ausgabe von LaTeX (oder welcher Prozessor auch immer durch `latex.engine()` gesetzt ist) statt von MathJax erstellen lässt. Diese Liste wird verwaltet durch die Befehle `latex.add_to_mathjax_avoid_list` und `latex.mathjax_avoid_list`.

```
sage: latex.mathjax_avoid_list([])
sage: latex.mathjax_avoid_list()
[]
sage: latex.mathjax_avoid_list(['foo', 'bar'])
sage: latex.mathjax_avoid_list()
['foo', 'bar']
sage: latex.add_to_mathjax_avoid_list('tikzpicture')
sage: latex.mathjax_avoid_list()
['foo', 'bar', 'tikzpicture']
sage: latex.mathjax_avoid_list([])
sage: latex.mathjax_avoid_list()
[]
```

Nehmen wir an ein LaTeX-Ausdruck wurde im Notebook durch `view()` oder während aktiviertem "Typeset" Knopf erzeugt. Und dann wird festgestellt, dass er die externe LaTeX-Installation benötigt, weil er in der `mathjax_avoid_list` steht. Der Ausdruck wird nun vom ausgewählten (durch `latex.engine()`) Prozessor erzeugt, und statt der Anzeige in einem externen Programm (was in der Kommandozeile passieren würde) wird

Sage versuchen das Ergebnis in einem einzigen, leicht beschnittenen Bild in der Ausgabezelle darzustellen.

Wie diese Umwandlung abläuft hängt von einigen Faktoren ab, hauptsächlich vom verwendeten LaTeX-Prozessor und davon welche Konvertierungswerkzeuge auf dem System vorhanden sind. Vier nützliche Konverter, die alle Eventualitäten abdecken sind `dvips`, `ps2pdf`, `dvipng` und aus dem `ImageMagick` Paket, `convert`. Das Ziel ist die Erzeugung einer .png Datei, die später wieder im Arbeitsblatt eingebunden werden kann. Wenn ein LaTeX-Ausdruck erfolgreich von `latex` in eine .dvi Datei verwandelt wird, dann sollte dvipng die Umwandlung vornehmen. Wenn der LaTeX Ausdruck und der gewählte LaTeX-Prozessor eine .dvi Datei mit Erweiterungen erstellt, die dvipng nicht unterstützt, so wird dvips eine PostScript-Datei erzeugen. So eine PostScript-Datei, oder eine .pdf Datei aus dem Prozessor `pdflatex`, wird dann von `convert` in eine .png Datei gewandelt. Das Vorhandensein von zweier solcher Konverter kann mit Hilfe der `have_dvipng()` und `have_convert()` Routinen überprüft werden.

Diese Umwandlungen werden automatisch ausgeführt, wenn Sie die nötigen Konverter installiert haben; falls nicht wird Ihnen eine Fehlermeldung angezeigt, die Ihnen sagt was fehlt und wo Sie es herunterladen können.

Für ein konkretes Beispiel wie komplizierte LaTeX-Ausdrücke verarbeitet werden können, sehen Sie sich das Beispiel des `tkz-graph` Pakets zum Erstellen von hochwertigen kombinatorischen Graphen im nächsten Abschnitt (*Ein Beispiel: Kombinatorische Graphen mit tkz-graph*) an. Für weitere Beispiele gibt es einige vorgepackte Testfälle. Um diese zu nutzen, müssen Sie das `sage.misc.latex.latex_examples` Objekt importieren. Dieses ist eine Instanz der `sage.misc.latex.LatexExamples` Klasse, wie unten beschrieben. Diese Klasse enthält momentan Beispiele von kommutativen Diagrammen, kombinatorischen Graphen, Knotentheorie und Beispiele für Graphen mit pstricks. Es werden damit die folgenden Pakete getestet: xy, tkz-graph, xypic, pstricks. Nach dem Import können Sie mittels Tab-Vervollständigung von `latex_examples` die vorgepackten Beispiele sehen. Bei Aufruf vom jedem Beispiel erhalten Sie eine Erklärung was nötig ist, damit das Beispiel korrekt dargestellt wird. Um die Darstellung tatsächlich zu sehen müssen Sie `view()` benutzen (sofern die Präambel, der LaTeX-Prozessor, etc richtig eingestellt sind).

```
sage: from sage.misc.latex import latex_examples
sage: latex_examples.diagram()
LaTeX example for testing display of a commutative diagram produced
by xypic.

To use, try to view this object -- it won't work.  Now try
'latex.add_to_preamble("\\usepackage[matrix,arrow,curve,cmtip]{xy}")',
and try viewing again -- it should work in the command line but not
from the notebook.  In the notebook, run
'latex.add_to_mathjax_avoid_list("xymatrix")' and try again -- you
should get a picture (a part of the diagram arising from a filtered
chain complex).
```

5.5 Ein Beispiel: Kombinatorische Graphen mit tkz-graph

Hochwertige Darstellungen von kombinatorischen Graphen (fortan nur noch "Graphen") sind mit Hilfe des `tkz-graph` Pakets möglich. Dieses Paket wurde ausbauend auf das `tikz` front-end der `pgf` Bibliothek entwickelt. Es müssen also all diese Komponenten Teil der systemweiten TeX-Installation sein, und es ist möglich, dass sie nicht in ihrer neusten Version in der TeX-Implementation vorliegen. Es ist also unter Umständen nötig oder ratsam diese Teile separat in Ihrem persönlichen texmf Baum zu installieren. Das Erstellen, Anpassen und Warten einer systemweiten oder persönlichen TeX-Installation würde allerdings den Rahmen dieses Dokuments sprengen. Es sollte aber einfach sein Anleitungen hierzu zu finden. Die nötigen Dateien sind unter *Eine vollfunktionsfähige TeX-Installation* aufgeführt.

Um also zu beginnen, müssen wir sicher sein, dass die relevanten Pakete eingefügt werden, indem wir sie in die Präambel des LaTeX-Dokuments hinzufügen. Die Bilder der Graphen werden nicht korrekt formatiert sein, wenn eine .dvi Datei als Zwischenergebnis erzeugt wird. Es ist also ratsam, den LaTeX-Prozessor auf `pdflatex` zu stellen.

Nun sollte ein Befehl wie `view(graphs.CompleteGraph(4))` in der Sage-Kommandozeile erfolgreich eine .pdf Datei mit einem Bild vom kompletten K_4 Graphen erzeugen.

Um das Gleiche im Notebook zu erstellen, müssen Sie MathJax für die Verarbeitung von LaTeX-Code ausschalten, indem Sie die "mathjax avoid list" benutzen. Graphen werden in einer `tikzpicture` Umgebung eingebunden, das ist also eine gute Wahl für die Zeichenkette für die Ausschlussliste. Jetzt sollte `view(graphs.CompleteGraph(4))` in einem Arbeitsblatt eine .pdf Datei mit pdflatex erstellen, mit dem `convert` Werkzeug eine .png Grafik erstellen und in die Ausgabezelle des Arbeitsblatts einfügen. Die folgenden Befehle veranschaulichen die Schritte einen Graphen mittels LaTeX in einem Notebook darzustellen.

```
sage: from sage.graphs.graph_latex import setup_latex_preamble
sage: setup_latex_preamble()
sage: latex.extra_preamble() # random - depends on system's TeX installation
'\\usepackage{tikz}\n\\usepackage{tkz-graph}\n\\usepackage{tkz-berge}\n'
sage: latex.engine('pdflatex')
sage: latex.add_to_mathjax_avoid_list('tikzpicture')
sage: latex.mathjax_avoid_list()
['tikzpicture']
```

Beachten Sie, dass es eine Vielzahl von Optionen gibt, die die Darstellung des Graphen in LaTeX mit `tkz-graph` beeinflussen. Auch das wiederrum ist nicht Ziel dieses Abschnitts. Sehen Sie sich hierfür den Abschnitt "LaTeX-Optionen für Graphen" aus dem Handbuch für weitere Anleitungen und Details an.

5.6 Eine vollfunktionsfähige TeX-Installation

Viele der erweiterten Integrationsmöglichkeiten von TeX in Sage benötigen eine systemweite Installation von TeX. Viele Linuxdistributionen bieten bereits TeX-Pakete basierend auf TeX-live, für OSX gibt es TeXshop und für Windows MikTeX. Das `convert` Werkzeug ist Teil der ImageMagick Suite (welche ein Paket oder zumindest ein simpler Download sein sollte). Die drei Programme `dvipng`, `ps2pdf`, und `dvips` sind wahrscheinlich bereits Teil Ihrer TeX Distribution. Die ersten beiden sollten auch von http://sourceforge.net/projects/dvipng/ als Teil von Ghostscript bezogen werden können.

Um kombinatorische Graphen darstellen zu können, wird eine aktuelle Version der PGF Bibliothek und die Dateien `tkz-graph.sty`, `tkz-arith.sty` und eventuell `tkz-berge.sty` benötigt, allesamt verfügbar auf der Altermundus Seite.

5.7 Externe Programme

Es sind drei Programme verfügbar um TeX weiter in Sage zu integrieren. Das erste ist sagetex. Eine kurze Beschreibung von sagetex wäre: Es ist eine Sammlung von TeX-Makros, die es einem LaTeX-Dokument erlauben Anweisungen einzubinden, mit denen Sage genutzt wird um verschiedene Objekte zu berechnen und/oder mittels eingebauter `latex()`-Funktion darzustellen. Als Zwischenschritt zum Kompilieren eines LaTeX-Dokuments werden also alle Berechnungs- oder LaTeX-Formatierungseigenschaften von Sage automatisch genutzt. Als Beispiel hierfür kann in einer mathematischen Betrachtung die korrekte Reihenfolge von Fragen und Antworten beibehalten werden, indem sagetex dazu genutzt wird Sage die einen aus den anderen berechnen zu lassen. Siehe hierfür auch *SageTeX nutzen*

tex2sws beginnt mit einem LaTeX-Dokument, aber definiert einige zusätzliche Umgebungen für Sage Code. Wenn es richtig genutzt wird, ist das Ergebnis ein Sage Arbeitsblatt mit korrekt von MathJax formatiertem Inhalt und dem dazugehörigen Sage Code in den Eingabezellen. Ein Lehrbuch oder Artikel kann also mit Sage Code Blöcken in LaTeX gesetzt werden und es kann "live" das ganze Dokument in ein Sage Arbeitsblatt überführt werden; unter Beibehaltung der Sage Code Blöcke und mit schön formatiertem mathematischen Text. Momentan in Arbeit, siehe tex2sws @ BitBucket .

sws2tex kehrt den Prozess um, indem es mit einem Sage Arbeitsblatt beginnt, und es in ein legitimes LaTeX-Dokument zur weiteren Bearbeitung mit allen LaTeX-Werkzeugen verwandelt. Momentan in Arbeit, siehe sws2tex @ BitBucket
.

KAPITEL 6

Programmierung

6.1 Sage-Dateien Laden und Anhängen

Als nächstes zeigen wir wie man Programme, die einer separaten Datei geschrieben wurden in Sage lädt. Erstellen Sie eine Datei, welche Sie `beispiel.sage` nennen mit folgendem Inhalt:

```
print "Hello World"
print 2^3
```

Sie können `beispiel.sage` einlesen und ausführen, indem Sie den `load`-Befehl verwenden.

```
sage: load "beispiel.sage"
Hello World
8
```

Sie können auch eine Sage-Datei an eine laufende Sitzung anhängen, indem Sie den `attach`-Befehl verwenden:

```
sage: attach "beispiel.sage"
Hello World
8
```

Wenn Sie nun `beispiel.sage` verändern und eine Leerzeile in Sage eingeben (d.h. `return` drücken) wird der Inhalt von `beispiel.sage` automatisch in Sage neu geladen.

Insbesondere lädt der `attach`-Befehl eine Datei jedesmal, wenn diese verändert wird automatisch neu, was beim Debuggen von Code nützlich sein kann, wobei der `load`-Befehl eine Datei nur einmal lädt.

Wenn Sage die Datei `beispiel.sage` lädt, wird sie zu Python-Code konvertiert, welcher dann vom Python-Interpreter ausgeführt wird. Diese Konvertierung ist geringfügig; sie besteht hautsächlich daraus Integer-Literale mit `Integer()` und Fließkomma-Literale mit `RealNumber()` zu versehen, ^ durch `**` zu ersetzen und z.B. `R.2` durch `R.gen(2)` auszutauschen. Die konvertierte Version von `beispiel.sage` befindet sich im gleichen Verzeichnis wie `beispiel.sage` und ist `beispiel.sage.py` genannt. Diese Datei enthält den folgenden Code:

```
print "Hello World"
print Integer(2)**Integer(3)
```

Integer-Literale wurden mit Integer() versehen und das ^ wurde durch ein `**` ersetzt. (In Python bedeutet ^ "exklusives oder" und `**` bedeutet "Exponentiation".)

Dieses "preparsing" ist in `sage/misc/interpreter.py` implementiert.)

Sie können mehrzeiligen eingerückten Code in Sage einfügen, solange Zeilenenden neue Blöcke kenntlich machen (dies ist in Dateien nicht notwendig). Jedoch beseht die beste Möglichkeit solchen Code in Sage einzufügen darin, diesen in einer Datei zu speichern und `attach` wie oben beschrieben zu verwenden.

6.2 Kompilierten Code erzeugen

Geschwindigkeit ist bei mathematischen Berechnungen äußerst wichtig. Python ist zwar eine komfortable Programmiersprache mit sehr hohen Abstraktionsniveau, jedoch können bestimmte Berechnungen mehrere Größenordnungen schneller als in Python sein, wenn sie in einer kompilierten Sprache mit statischen Datentypen implementiert wurden. Manche Teile von Sage würden zu langsam sein, wenn sie komplett in Python geschrieben wären. Um dies zu berücksichtigen unterstützt Sage eine kompilierte "Version" von Python, welche Cython ([Cyt] und [Pyr]) genannt wird. Cython ist gleichzeitig sowohl zu Python, als auch zu C ähnlich. Die meisten von Pythons Konstruktionen, einschließlich "list comprehensions", bedingte Ausdrücke und Code wie += sind erlaubt; Sie können auch Code importieren, den Sie in anderen Python-Modulen geschrieben haben. Darüberhinaus können Sie beliebige C Variablen definieren und beliebe C-Bibliothekaufrufe direkt ausführen. Der daraus entstehende Code wird nach C konvertiert und mithilfe eines C-Compilers kompiliert.

Um eigenen kompilierten Sagecode zu erstellen, geben Sie der Datei eine `.spyx` Endung (anstelle von `.sage`). Falls Sie mit der Kommandozeile arbeiten, können Sie kompilierten Code genau wie interpretierten Code anhängen und laden. (Im Moment wird das Anhängen von Cythoncode vom Notebook aus nicht unterstützt). Die tatsächliche Kompilierung wird "hinter den Kulissen" durchgeführt ohne dass Sie explizit etwas tun müssen. Schauen Sie sich `$SAGE_ROOT/examples/programming/sagex/factorial.spyx` an, um ein Beispiel einer kompilierten Implementation der Fakultätsfunktion zu sehen, welche die GMP-C-Bibliothek unmittelbar benutzt. Um dies selbst auszuprobierten, wechseln Sie in das Verzeichnis `$SAGE_ROOT/examples/programming/sagex/` und führen Sie Folgendes aus:

```
sage: load "factorial.spyx"
***************************************************
                Recompiling factorial.spyx
***************************************************
sage: factorial(50)
30414093201713378043612608166064768844377641568960512000000000000L
sage: time n = factorial(10000)
CPU times: user 0.03 s, sys: 0.00 s, total: 0.03 s
Wall time: 0.03
```

Hier gibt das abschließende L ein "Python long integer" zu erkennen (Lesen Sie *Der Pre-Parser: Unterschiede zwischen Sage und Python*).

Beachten Sie, dass Sage `factorial.spyx` neu kompiliert falls Sie Sage beenden und neustarten. Die komplierte "shared object library" wird unter `$HOME/.sage/temp/hostname/pid/spyx` gespeichert. Diese Dateien werden gelöscht wenn Sie Sage beenden.

Auf spyx-Dateien wird kein "preparsing" angewendet, d.h. `1/3` wird in einer spyx-Datei zu 0 ausgewertet, anstelle der rationalen Zahl $1/3$. Wenn `foo` eine Funktion in der Sage-Bibliothek ist die Sie verwenden möchten, müssen Sie `sage.all` importieren und `sage.all.foo` benutzen.

```
import sage.all
def foo(n):
    return sage.all.factorial(n)
```

6.2.1 Auf C-Funktionen in separaten Dateien zugreifen

Es ist auch nicht schwer auf C-Funktions zuzugreifen welche in separaten *.c Dateien definiert sind. Hier ist ein Beispiel. Erzeugen Sie die Dateien `test.c` und `test.spyx` in dem gleichen Verzeichnis mit den Inhalten:

Der reine C-Code: `test.c`

```
int add_one(int n) {
  return n + 1;
}
```

Der Cython-Code: `test.spyx`:

```
cdef extern from "test.c":
    int add_one(int n)

def test(n):
    return add_one(n)
```

Dann funktioniert das Folgende:

```
sage: attach "test.spyx"
Compiling (...)/test.spyx...
sage: test(10)
11
```

Wenn die zusätzliche Bibliothek `foo` gebraucht wird um den C-Code, der aus einer Cython-Datei generiert wurde zu kompilieren, fügen Sie die Zeile `clib foo` zu dem Cython-Quellcode hinzu. Auf ähnliche Weise kann eine zusätzliche Datei `bar` zu der Kompilierung mit der Deklaration `cfile bar` hinzugefügt werden.

6.3 eigenständige Python/Sage Skripte

Das folgende eigenständige Sageskript faktorisiert ganze Zahlen, Polynome, usw.:

```
#!/usr/bin/env sage -python

import sys
from sage.all import *

if len(sys.argv) != 2:
    print "Usage: %s <n>"%sys.argv[0]
    print "Outputs the prime factorization of n."
    sys.exit(1)

print factor(sage_eval(sys.argv[1]))
```

Um dieses Skript benutzen zu können muss `SAGE_ROOT` in ihrer PATH-Umgebungsvariable enthalten sein. Falls das das obige Skript `factor` genannt wurde, ist hier ein beispielhafter Aufruf:

```
bash $ ./factor 2006
2 * 17 * 59
bash $ ./factor "32*x^5-1"
(2*x - 1) * (16*x^4 + 8*x^3 + 4*x^2 + 2*x + 1)
```

6.4 Datentypen

Jedes Objekt hat in Sage einen wohldefinierten Datentyp. Python besitzt eine Vielzahl von standardmäßiger elementarer Datentypen und die Sage-Bibliothek fügt noch viele weitere hinzu. Zu Pythons standardmäßigen Datentypen gehören Strings, Listen, Tupel, Ganzzahlen und Gleitkommazahlen, wie hier zu sehen ist:

```
sage: s = "sage"; type(s)
<type 'str'>
sage: s = 'sage'; type(s)      # Sie können einfache oder doppelte Anführungszeichen verwenden
<type 'str'>
sage: s = [1,2,3,4]; type(s)
<type 'list'>
sage: s = (1,2,3,4); type(s)
<type 'tuple'>
sage: s = int(2006); type(s)
<type 'int'>
sage: s = float(2006); type(s)
<type 'float'>
```

Hierzu fügt Sage noch viele weitere hinzu. Zum Beispiel Vektorräume:

```
sage: V = VectorSpace(QQ, 1000000); V
Vector space of dimension 1000000 over Rational Field
sage: type(V)
<class 'sage.modules.free_module.FreeModule_ambient_field_with_category'>
```

Nur bestimmte Funktionen können auf `V` aufgerufen werden. In anderen mathematischen Softwaresystemem würde dies mit der "Funktionalen"-Notation `foo(V,...)` geschehen. In Sage sind bestimmte Funktionen an den Typ (oder der Klasse) von `V` angehängt, und diese werden unter Benutzung einer objektorientierten Syntax, wie in Java oder C++ aufgerufen. Zum Beispiel `V.foo(...)`. Dies hilft dabei eine Überfüllung des globalen Namensraums mit tausenden von Funktionen zu vermeiden. Das bedeutet auch, dass viele verschiedene Funktionen mit unterschiedlichen Funktionsweisen foo genannt werden können, ohne dass der Typ des Arguments überprüft (oder Case-Anweisungen ausgeführt) werden muss, um zu entscheiden welche aufgerufen werden soll. Weiterhin ist die Funktion auch dann noch verfügbar, wenn ihr Name zu einem anderen Zweck verwendet wurde. (Zum Beispiel wenn Sie etwas `zeta` nennen und dann den Wert der Riemannschen Zeta-Funktion bei 0.5 berechnen wollen, können Sie immernoch `s=.5; s.zeta()` benutzen).

```
sage: zeta = -1
sage: s=.5; s.zeta()
-1.46035450880959
```

In manchen sehr oft auftretenden Fällen wird auch die gewöhnliche funktionale Notation unterstützt, da dies bequem ist und manche mathematische Ausdrücke in objektorientierter Notation verwirrend aussehen könnten. Hier sind einige Beispiele:

```
sage: n = 2; n.sqrt()
sqrt(2)
sage: sqrt(2)
sqrt(2)
sage: V = VectorSpace(QQ,2)
sage: V.basis()
    [
    (1, 0),
    (0, 1)
    ]
sage: basis(V)
    [
    (1, 0),
    (0, 1)
    ]
sage: M = MatrixSpace(GF(7), 2); M
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 7
sage: A = M([1,2,3,4]); A
[1 2]
```

```
[3 4]
sage: A.charpoly('x')
x^2 + 2*x + 5
sage: charpoly(A, 'x')
x^2 + 2*x + 5
```

Um alle Member-Funktionen von A anzuzeigen, können Sie die Tab-Vervollständigung benutzen. Tippen Sie einfach `A.`, dann die `[tab]`-Taste auf Ihrer Tastatur, wie es in *Rückwärtssuche und Tab-Vervollständigung* beschrieben ist.

6.5 Listen, Tupel, und Folgen

Der Listen-Datentyp speichert Elemente eines beliebigen Typs. Wie in C, C++, usw. (jedoch anders als in vielen gewöhnlichen Computer-Algebra-Systemen), die Elemente der Liste werden bei 0 beginnend indiziert:

```
sage: v = [2, 3, 5, 'x', SymmetricGroup(3)]; v
[2, 3, 5, 'x', Symmetric group of order 3! as a permutation group]
sage: type(v)
<type 'list'>
sage: v[0]
2
sage: v[2]
5
```

(Wenn man auf ein Listenelement zugreift ist es OK wenn der Index kein Python int ist!) Mit einem Sage-Integer (oder Rational, oder mit allem anderen mit einer `__index__` Methode) funktioniert es genauso.

```
sage: v = [1,2,3]
sage: v[2]
3
sage: n = 2      # SAGE Integer
sage: v[n]       # Perfectly OK!
3
sage: v[int(n)]  # Also OK.
3
```

Die `range`-Funktion erzeugt eine Liste von Python int's (nicht Sage-Integers):

```
sage: range(1, 15)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Dies ist nützlich wenn man List-Comprehensions verwendet um Listen zu konstruieren:

```
sage: L = [factor(n) for n in range(1, 15)]
sage: print L
[1, 2, 3, 2^2, 5, 2 * 3, 7, 2^3, 3^2, 2 * 5, 11, 2^2 * 3, 13, 2 * 7]
sage: L[12]
13
sage: type(L[12])
 <class 'sage.structure.factorization_integer.IntegerFactorization'>
sage: [factor(n) for n in range(1, 15) if is_odd(n)]
[1, 3, 5, 7, 3^2, 11, 13]
```

Um mehr darüber zu erfahren wie man Listen mit Hilfe von List-Comprehensions erzeugt, lesen Sie [PyT].

List-Slicing ist eine wunderbare Eigenschaft. Wenn `L` eine Liste ist, dann gibt `L[m:n]` die Teilliste von `L` zurück, die erhalten wird wenn man mit dem m^{ten} Element beginnt und bei dem $(n-1)^{ten}$ Element aufhört, wie unten gezeigt wird.

```
sage: L = [factor(n) for n in range(1, 20)]
sage: L[4:9]
[5, 2 * 3, 7, 2^3, 3^2]
sage: print L[:4]
[1, 2, 3, 2^2]
sage: L[14:4]
[]
sage: L[14:]
[3 * 5, 2^4, 17, 2 * 3^2, 19]
```

Tupel sind ähnlich wie Listen, außer dass sie unveränderbar sind, was bedeutet dass sie, sobald sie erzeugt wurden, nicht mehr verändert werden können.

```
sage: v = (1,2,3,4); v
(1, 2, 3, 4)
sage: type(v)
<type 'tuple'>
sage: v[1] = 5
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

Folgen sind ein dritter an Listen angelehnter Sage-Datentyp. Anders als Listen und Tupel, sind Folgen kein gewöhnlicher Python-Datentyp. Standardmäßig sind Folgen veränderbar, mit der `Sequence`-Klassenmethode `set_immutable` können sie auf unveränderbar gestellt werden, wie das folgende Beispiel zeigt. Alle Elemente einer Folge haben einen gemeinsamen Obertyp, der das Folgenuniversum genannt wird.

```
sage: v = Sequence([1,2,3,4/5])
sage: v
[1, 2, 3, 4/5]
sage: type(v)
<class 'sage.structure.sequence.Sequence_generic'>
sage: type(v[1])
<type 'sage.rings.rational.Rational'>
sage: v.universe()
Rational Field
sage: v.is_immutable()
False
sage: v.set_immutable()
sage: v[0] = 3
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Folgen sind von Listen abgeleitet und können überall dort verwendet werden, wo auch Listen benutzt werden können.

```
sage: v = Sequence([1,2,3,4/5])
sage: isinstance(v, list)
True
sage: list(v)
[1, 2, 3, 4/5]
sage: type(list(v))
<type 'list'>
```

Ein weiteres Beispiel von unveränderbaren Folgen sind Basen von Vektorräumen. Es ist wichtig, dass sie nicht verändert werden können.

```
sage: V = QQ^3; B = V.basis(); B
[
```

```
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: type(B)
<class 'sage.structure.sequence.Sequence_generic'>
sage: B[0] = B[1]
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: B.universe()
Vector space of dimension 3 over Rational Field
```

6.6 Dictionaries

Ein Dictionary (manchmal auch assoziativer Array genannt) ist eine Abbildung von 'hashbaren' Objekten (z.B. Strings, Zahlen und Tupel; Lesen Sie die Python documentation http://docs.python.org/tut/node7.html und http://docs.python.org/lib/typesmapping.html für weitere Details) zu beliebigen Objekten.

```
sage: d = {1:5, 'sage':17, ZZ:GF(7)}
sage: type(d)
<type 'dict'>
sage: d.keys()
 [1, 'sage', Integer Ring]
sage: d['sage']
17
sage: d[ZZ]
Finite Field of size 7
sage: d[1]
5
```

Der dritte "key" zeigt, dass Indizes eines Dictionaries kompliziert, also beispielsweise der Ring der ganzen Zahlen, sein können.

Sie können das obige Dictionary auch in eine Liste mit den gleichen Daten umwandeln:

```
sage: d.items()
[(1, 5), ('sage', 17), (Integer Ring, Finite Field of size 7)]
```

Eine häufig vorkommende Ausdrucksweise ist über einem Paar in einem Dictionary zu iterieren:

```
sage: d = {2:4, 4:16, 3:9}
sage: [a*b for a, b in d.iteritems()]
[8, 27, 64]
```

Ein Dictionary ist ungeordnet, wie die letzte Ausgabe verdeutlicht.

6.7 Mengen

Python hat einen standardmäßigen Mengen-Datentyp. Sein Hauptmerkmal ist, neben weiteren typischen Mengenoperationen, dass das Nachschlagen ob ein Element zu der Menge gehört oder nicht, sehr schnell geht.

```
sage: X = set([1,19,'a']);   Y = set([1,1,1, 2/3])
sage: X
set(['a', 1, 19])
```

```
sage: Y
set([1, 2/3])
sage: 'a' in X
True
sage: 'a' in Y
False
sage: X.intersection(Y)
set([1])
```

Sage besitzt auch einen eigenen Mengen-Datentyp, welcher (manchmal) mit Hilfe des standardmäßigen Python-Mengen-Datentyps implementiert ist, jedoch darüberhinaus manche Sage-spezifischen Funktionen aufweist. Sie können eine Sage-Menge erzeugen indem Sie `Set(...)` verwenden. Zum Beispiel,

```
sage: X = Set([1,19,'a']);   Y = Set([1,1,1, 2/3])
sage: X
{'a', 1, 19}
sage: Y
{1, 2/3}
sage: X.intersection(Y)
{1}
sage: print latex(Y)
\left\{1, \frac{2}{3}\right\}
sage: Set(ZZ)
Set of elements of Integer Ring
```

6.8 Iteratoren

Iteratoren sind seit Version 2.2 ein Teil von Python und erweisen sich in mathematischen Anwendungen als besonders nützlich. Wir geben hier ein paar Beispiele an; Lesen Sie [PyT] um weitere Details zu erfahren. Wir erstellen einen Iterator über die Quadrate der nichtnegativen ganzen Zahlen bis 10000000.

```
sage: v = (n^2 for n in xrange(10000000))
sage: v.next()
0
sage: v.next()
1
sage: v.next()
4
```

Nun erzeugen wir einen Iterator über den Primzahlen der Form $4p + 1$ wobei auch p prim ist und schauen uns die ersten Werte an.

```
sage: w = (4*p + 1 for p in Primes() if is_prime(4*p+1))
sage: w         # in the next line, 0xb0853d6c is a random 0x number
<generator object at 0xb0853d6c>
sage: w.next()
13
sage: w.next()
29
sage: w.next()
53
```

Bestimmte Ringe, z. B. endliche Körper und die ganzen Zahlen, haben zugehörige Iteratoren:

```
sage: [x for x in GF(7)]
[0, 1, 2, 3, 4, 5, 6]
sage: W = ((x,y) for x in ZZ for y in ZZ)
```

```
sage: W.next()
(0, 0)
sage: W.next()
(0, 1)
sage: W.next()
(0, -1)
```

6.9 Schleifen, Funktionen, Kontrollstrukturen und Vergleiche

Wir haben schon ein paar Beispiele gesehen in denen die `for`-Schleife üblicherweise Verwendung findet. In Python hat eine `for`-Schleife eine eingerückte Struktur, wie hier:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Beachten Sie den Doppelpunkt am Ende der for-Anweisung (dort befindet sich kein "do" oder "od" wie in GAP oder Maple) und die Einrückung vor dem Schleifenrumpf, dem `print(i)`. Diese Einrückung ist wichtig. In Sage wird die Einrückung automatisch hinzugefügt wenn Sie nach einem ":" die `enter`-Taste drücken, wie etwa im Folgenden Beispiel.

```
sage: for i in range(5):
....:     print(i)  # now hit enter twice
....:
0
1
2
3
4
```

Das Symbol = wird bei Zuweisungen verwendet. Das Symbol == wird verwendet um Gleichheit zu testen:

```
sage: for i in range(15):
...       if gcd(i,15) == 1:
...           print(i)
1
2
4
7
8
11
13
14
```

Behalten Sie im Gedächtnis, dass die Block-Struktur von `if`, `for` und `while` Ausdrücken durch die Einrückung bestimmt wird:

```
sage: def legendre(a,p):
...       is_sqr_modp=-1
...       for i in range(p):
...           if a % p == i^2 % p:
...               is_sqr_modp=1
```

```
...         return is_sqr_modp

sage: legendre(2,7)
1
sage: legendre(3,7)
-1
```

Natürlich ist dies keine effiziente Implementierung des Legendre-Symbols! Dies soll nur bestimmte Aspekte won Python/Sage verdeutlichen. Die Funktion {kronecker}, welche zu Sage gehört, berechnet das Legendre-Symbol effizient mittels eines Aufrufs von PARIs C-Bibliothek.

Schließlich merken wir an, dass Vergleiche wie ==, !=, <=, >=, >, < von zwei Zahlen automatisch beide Zahlen in den gleichen Typ konvertieren, falls dies möglich ist:

```
sage: 2 < 3.1; 3.1 <= 1
True
False
sage: 2/3 < 3/2;   3/2 < 3/1
True
True
```

Fast immer können zwei beliebige Objekte verglichen werden. Es gibt keine Voraussetzung die besagt, dass die Objekte mit einer totalen Ordnung versehen sein müssen.

```
sage: 2 < CC(3.1,1)
True
sage: 5 < VectorSpace(QQ,3)    # output can be somewhat random
True
```

Nutzen Sie bool für symbolische Ungleichungen:

```
sage: x < x + 1
x < x + 1
sage: bool(x < x + 1)
True
```

Beim Vergleichen von Objekten unterschiedlichen Typs versucht Sage in den meisten Fällen eine kanonische Umwandlung beider Objekte in einen gemeinsamen Typ zu finden. Falls erfolgreich wird der Vergleich auf den umgewandelten Objekten durchgeführt; Falls nicht erfolgreich werden die Objekte als ungleich angesehen. Um zu Testen, ob zwei Variablen auf das gleiche Objekt zeigen, verwenden Sie `is`. Zum Beispiel:

```
sage: 1 is 2/2
False
sage: 1 is 1
False
sage: 1 == 2/2
True
```

In den folgenden zwei Zeilen ist der erste Gleichheitstest `False`, da es keinen kanonischen Morphismus $\mathbf{Q} \to \mathbf{F}_5$ gibt, also gibt es keine kanonische Möglichkeit die 1 in $\mathbf{F}_5$ mit der $1 \in \mathbf{Q}$ zu vergleichen. Im Gegensatz dazu gibt es eine kanonische Abbildung $\mathbf{Z} \to \mathbf{F}_5$, also ist der zweite Gleichheitstest `True`. Beachten Sie auch, dass die Reihenfolge keine Rolle spielt.

```
sage: GF(5)(1) == QQ(1); QQ(1) == GF(5)(1)
False
False
sage: GF(5)(1) == ZZ(1); ZZ(1) == GF(5)(1)
True
True
```

```
sage: ZZ(1) == QQ(1)
True
```

WARNUNG: Vergleiche in Sage sind restriktiver als in Magma, welches die $1 \in \mathbf{F}_5$ gleich der $1 \in \mathbf{Q}$ festlegt.

```
sage: magma('GF(5)!1 eq Rationals()!1')            # optional - magma
true
```

6.10 Profiling

Autor des Abschnitts: Martin Albrecht (malb@informatik.uni-bremen.de)

> "Premature optimization is the root of all evil." - Donald Knuth

Manchmal ist es nützlich nach Engstellen im Code zu suchen, um zu verstehen welche Abschnitte die meiste Berechnungszeit beanspruchen; dies kann ein guter Hinweis darauf sein, welche Teile optimiert werden sollten. Python, und daher auch Sage, stellen mehrere "Profiling" – so wird dieser Prozess genannt – Optionen zur Verfügung.

Am einfachsten zu Benutzen ist das `prun`-Kommando in der interaktiven Shell. Es gibt eine Zusammenfassung zurück, die beschreibt welche Funktionen wie viel Berechnungszeit veranschlagt haben. Um die (zu diesem Zeitpunkt langsame) Matrixmultiplikation über endlichen Körpern zu Profilieren, geben Sie z.B. folgendes ein:

```
sage: k,a = GF(2**8, 'a').objgen()
sage: A = Matrix(k,10,10,[k.random_element() for _ in range(10*10)])

sage: %prun B = A*A
       32893 function calls in 1.100 CPU seconds

Ordered by: internal time

ncalls tottime percall cumtime percall filename:lineno(function)
 12127  0.160   0.000   0.160  0.000 :0(isinstance)
  2000  0.150   0.000   0.280  0.000 matrix.py:2235(__getitem__)
  1000  0.120   0.000   0.370  0.000 finite_field_element.py:392(__mul__)
  1903  0.120   0.000   0.200  0.000 finite_field_element.py:47(__init__)
  1900  0.090   0.000   0.220  0.000 finite_field_element.py:376(__compat)
   900  0.080   0.000   0.260  0.000 finite_field_element.py:380(__add__)
     1  0.070   0.070   1.100  1.100 matrix.py:864(__mul__)
  2105  0.070   0.000   0.070  0.000 matrix.py:282(ncols)
  ...
```

Hier ist `ncalls` die Anzahl der Aufrufe, `tottime` ist die Gesamtzeit, die für die Funktion verwendet wurde (ausgenommen der Zeit, die für Unterfunktionsaufrufe verwendet wurde), `percall` ist der Quotient von `tottime` geteilt durch `ncalls`. `cumtime` ist die Gesamtzeit, die für diese Funktion und alle Unterfunktionsaufrufe (d.h., vom Aufruf bis zum Ende) verwendet wurde, `percall` ist der Quotient von `cumtime` geteilt durch die Zeit elementarer Funktionsaufrufe, und `filename:lineno(function)` stellt die entsprechenden Daten jeder Funktion zur Verfügung. Die Daumenregel ist hier: Je höher die Funktion in dieser Liste steht, desto teurer ist sie. Also ist sie interessanter für Optimierungen.

Wie sonst auch stellt `prun?` Details zur Benutzung des Profilers und zum Verstehen seines Outputs zur Verfügung.

Die Profilierungsdaten können auch in ein Objekt geschrieben werden um eine weitere Untersuchung zu ermöglichen:

```
sage: %prun -r A*A
sage: stats = _
sage: stats?
```

Beachten Sie: das Eingeben von `stats = prun -r A\*A` erzeugt eine Syntaxfehlermeldung, da prun ein IPython-Shell-Kommando ist und keine reguläre Funktion.

Um eine schöne graphische Repräsentation der Profilerdaten zu erhalten, können Sie den "hotshot-Profiler", ein kleines Skript genannt `hotshot2cachetree` und das Programm `kcachegrind` (nur für Unix) benutzen. Hier ist das gleiche Beispiel mit dem "hotshot-Profiler":

```
sage: k,a = GF(2**8, 'a').objgen()
sage: A = Matrix(k,10,10,[k.random_element() for _ in range(10*10)])
sage: import hotshot
sage: filename = "pythongrind.prof"
sage: prof = hotshot.Profile(filename, lineevents=1)

sage: prof.run("A*A")
<hotshot.Profile instance at 0x414c11ec>
sage: prof.close()
```

Dies führt zu einer Datei `pythongrind.prof` in aktuellen Datenverzeichnis. Diese kann nun zur Visualisierung in das cachegrind-Format konvertiert werden.

Tippen Sie in einer System-Shell:

```
hotshot2calltree -o cachegrind.out.42 pythongrind.prof
```

Die Ausgabedatei `cachegrind.out.42` kann nun mit `kcachegrind` untersucht werden. Bitte beachten Sie, dass die Namenskonvention `cachegrind.out.XX` erhalten bleiben muss.

SageTeX nutzen

Das SageTeX Paket ermöglicht es Ihnen die Ergebnisse von Sage Berechnungen direkt in ein LaTeX-Dokument zu setzen. Es wird standardmäßig mit Sage installiert. Um es zu nutzen müssen Sie es lediglich in Ihrem lokalen TeX-System "installieren", wobei "installieren" hier eine einzige Datei kopieren bedeutet. Siehe hierfür auch *Installation* in diesem Tutorial und den Abschnitt "Make SageTeX known to TeX" des Sage installation guide (dieser Link sollte Sie zu einer lokalen Kopie der Installationsanleitung führen) um weitere Informationen zu erhalten.

Hier stellen wir ein sehr kurzes Beispiel vor wie man SageTeX nutzt. Die komplette Dokumentation finden Sie unter `SAGE_ROOT/local/share/texmf/tex/generic/sagetex`, wobei `SAGE_ROOT` das Installationsverzeichnis von Sage ist. Dieses Verzeichnis enthält die Dokumentation, eine Beispieldatei und einige nützliche Python Skripte.

Um zu sehen wie SageTeX funktioniert, folgen Sie den Anweisungen zur Installation von SageTeX (in *Installation*) und kopieren Sie den folgenden Text in eine Datei namens - sagen wir `st_example.tex`:

> **Warnung:** Der folgende Text wird mehrere Fehler bezüglich unbekannten Kontrollsequenzen anzeigen, wenn Sie ihn in der "live" Hilfe ansehen. Nutzen Sie stattdessen die statische Version um den korrekten Text zu sehen.

```
\documentclass{article}
\usepackage{sagetex}

\begin{document}

Wenn Sie Sage\TeX nutzen, können Sie Sage nutzen um Dinge auszurechen und
sie direkt in ein \LaTeX{} Dokument zu setzen. Zum Beispiel gibt es
$\sage{number_of_partitions(1269)}$ ganzzahlige Partitionen von $1269$.
Sie müssen die Zahl nicht selbst ausrechnen, oder aus einem anderen
Programm herauskopieren.

Hier ein wenig Sage Code:

\begin{sageblock}
    f(x) = exp(x) * sin(2*x)
\end{sageblock}

Die zweite Ableitung von $f$ ist

\[
  \frac{\mathrm{d}^{2}}{\mathrm{d}x^{2}} \sage{f(x)} =
  \sage{diff(f, x, 2)(x)}.
```

```
\]

Hier ein Plot von $f$ von $-1$ bis $1$:

\sageplot{plot(f, -1, 1)}

\end{document}
```

Lassen Sie LaTeX ganz normal über `st_example.tex` laufen. Beachten Sie dabei, dass LaTeX sich über einige Dinge beschwert, z.B.:

```
Package sagetex Warning: Graphics file
sage-plots-for-st_example.tex/plot-0.eps on page 1 does not exist. Plot
command is on input line 25.

Package sagetex Warning: There were undefined Sage formulas and/or
plots. Run Sage on st_example.sage, and then run LaTeX on
st_example.tex again.
```

Beachten Sie, dass zusätzlich zu den Dateien, die LaTeX normalerweise produziert noch eine Datei `st_example.sage` erscheint. Das ist das Sage Skript, das erstellt wurde als Sie LaTeX mit `st_example.tex` aufgerufen haben. Wie Ihnen die Warnmeldung mitteilte sollten Sie Sage über die Datei `st_example.sage` laufen lassen, also tun Sie das bitte. Ihnen wird gesagt werden, dass Sie LaTeX erneut über die Datei `st_example.tex` laufen lassen sollen; bevor Sie dies tun beachten Sie, dass eine neue Datei namens `st_example.sout` von Sage erstellt wurde. Diese Datei enthält die Ergebnisse von Sages Berechnungen in einem Format, das LaTeX nutzen kann um es in Ihren Text einzufügen. Ein neues Verzeichnis mit einer .eps Datei Ihres Plots wurde ebenfalls erstellt. Lassen Sie LaTeX nun erneut laufen, und Sie werden sehen, dass alles was Sage berechnet und geplottet hat nun in Ihrem Dokument erscheint.

Die verschiednenen verwendeten Makros sollten einfach zu verstehen sein. Eine `sageblock` Umgebung setzt Ihren Code unverändert und führt ihn auch aus wenn Sie Sage laufen lassen. Wenn Sie etwa `\sage{foo}` schreiben, wird das Ergebnis des Aufrufs `latex(foo)` (in Sage) in Ihrem Dokument erscheinen. Plot-Befehle sind etwas komplizierter, aber in Ihrer einfachsten Form fügt `\sageplot{foo}` das Bild ein, das Sie erhalten wenn Sie `foo.save('filename.eps')` in Sage aufrufen würden.

Grundsätzlich gilt:

- lassen Sie LaTeX über Ihre .tex Datei laufen;
- lassen Sie Sage über die neu generierte .sage Datei laufen;
- lassen Sie LaTeX erneut laufen.

Sie können das Aufrufen von Sage weglassen, wenn Sie keine Änderung an den Sage Befehlen in Ihrem Dokument vorgenommen haben.

Es gibt noch viel mehr über SageTeX zu sagen, aber da sowohl Sage alsauch LaTeX komplexe und mächtige Werkzeuge sind, sollten Sie die Dokumentation über SageTeX in `SAGE_ROOT/local/share/texmf/tex/generic/sagetex` lesen.

KAPITEL 8

Nachwort

8.1 Warum Python?

8.1.1 Vorteile von Python

Sage ist hautsächlich in der Programmiersprache Python implementiert (siehe [Py]). Jedoch ist Code, bei dem Geschwindigkeit ausschlaggebend ist, in einer kompilierten Sprache implementiert. Python hat folgende Vorteile:

- **Speichern von Objekten** wird in Python gut unterstützt. Für das Speichern von (nahezu) beliebigen Objekten auf Festplatten oder in Datenbanken sind in Python weitgehende Hilfsmittel vorhanden.
- Exzellente Unterstützung für die **Dokumentation** von Funktionen und Paketen im Quellcode, einschließlich der automatischen Erstellung der Dokumentation und automatisches Testen aller Beispiele. Die Beispiele werden regelmäßig automatisch getestet und es wird garantiert, dass sie wie angegeben funktionieren.
- **Speicherverwaltung**: Python besitzt nun einen gut durchdachten und robusten Speicherverwalter und einen Speicherbereiniger, der zirkuläre Referenzen korrekt behandelt und lokale Variablen in Dateien berücksichtigt.
- Python besitzt mittlerweile **viele Pakete**, die für Sagenutzer sehr reizvoll sein könnten: numerische Analysis und lineare Algebra, 2D und 3D Visualisierungen, Vernetzungen (für verteilte Berechnungen und Server, z.B. mithilfe von twisted), Datenbankunterstützung, usw.
- **Portabilität:** Python kann auf den meisten Systemen unkompliziert, innerhalb von Minuten aus dem Quellcode kompiliert werden.
- **Fehlerbehandlung:** Python besitzt ein ausgeklügeltes und wohl durchdachtes System für die Behandlung von Ausnahmebedingunen, mit dem Programme sinnvoll weiterarbeiten können, sogar wenn bei ihrem Aufruf Fehler auftreten.
- **Debugger:** Python beinhaltet einen Debugger. Folglich kann der Benutzer, falls der Code aus irgendeinem Grund fehlschlägt, auf eine ausgiebige Stack-Ablaufverfolgung zugreifen, den Zustand aller relevanter Variablen betrachten, und sich auf dem Stack nach oben oder unten bewegen.
- **Profiler:** Es gibt einen Python-Profiler, welcher Code ausführt und einen Bericht erstellt, in dem detailliert aufgestellt wurde wie oft und wie lange jede Funkion aufgerufen wurde.
- **Eine Sprache:** Anstatt eine **neue Sprache** für mathematische Software zu schreiben, wie es für Magma, Maple, Mathematica, Matlab, GP/PARI, GAP, Macaulay 2, Simath, usw. gemacht wurde, benutzen wir die Programmiersprache Python, eine beliebte Programmiersprache, die von hunderten begabten Softwareingenieuren rege weiterentwickelt und optimiert wird. Python ist eine bedeutete Open-Source Erfolgsgeschichte mit einem ausgereiften Entwicklungsprozess. (siehe [PyDev]).

8.1.2 Der Pre-Parser: Unterschiede zwischen Sage und Python

Aus mathematischer Sicht kann Python in verschiedener Weise verwirrend sein, also verhält sich Sage an manchen Stellen anders als Python.

- **Notation für Exponentiation:** `**` versus `^`. In Python bedeutet `^` "xor", und nicht Exponentiation, also gilt in Python:

  ```
  >>> 2^8
  10
  >>> 3^2
  1
  >>> 3**2
  9
  ```

  Diese Benutzung von `^` kann merkwürdig erscheinen und sie ist ineffizient für mathematische Anwender, da die "Exklusives-Oder"-Funktion nur selten verwendet wird. Um dies zu beheben parst Sage alle Kommandozeilen bevor es diese zu Python weitergibt und ersetzt jedes Auftreten `^`, das in keinem String vorkommt mit `**`:

  ```
  sage: 2^8
  256
  sage: 3^2
  9
  sage: "3^2"
  '3^2'
  ```

- **Integerdivision:** Der Pythonaudruck `2/3` verhält sich nicht so, wie es Mathematiker erwarten würden. In Python wird, falls `m` und `n` Integer sind, auch `m/n` als Integer behandelt, es ist nämlich der Quotient von `m` geteilt durch `n`. Daher ist `2/3=0`. Es wurde in der Pythoncommunity darüber geredet, ob in Python die Division geändert werden sollte, so dass `2/3` die Gleitkommazahl `0.6666...` zurückgibt und `2//3` das Ergebnis `0` hat.

  Wir berücksichtigen dies im Sage-Interpreter indem wir Integer-Literale mit `Integer( )` versehen und die Division als Konstruktor für rationale Zahlen behandeln. Zum Beispiel:

  ```
  sage: 2/3
  2/3
  sage: (2/3).parent()
  Rational Field
  sage: 2//3
  0
  sage: int(2)/int(3)
  0
  ```

- **Große ganze Zahlen:** Python besitzt von Hause aus Unterstützung für beliebig große ganze Zahlen zusätzlich zu C-ints. Diese sind bedeutend langsamer als die von GMP zur Verfügung gestellten und sie haben die Eigenschaft, dass die mit einem `L` am Ende ausgegeben werden um sie von ints unterscheiden zu können (und dies wird sich in naher Zeit nicht ändern). Sage implementiert beliebig große Integers mit Hilfe der GMP C-Bibliothek, und diese werden ohne `L` ausgegeben.

Anstatt den Python-Interpreter zu verändern (wie es mache Leute für interne Projekte getan haben), benutzen wir die Sprache Python unverändert und haben einen Prä-Parser geschrieben, so dass sich die Kommandozeilen-IPython-Version so verhält, wie es Mathematiker erwarten würden. Dies bedeutet, dass bereits existierender Python-Code in Sage so verwendet werden kann wie er ist. Man muss jedoch immernoch die standardmäßigen Python-Regeln beachten, wenn man Pakete schreibt, die in Sage importiert werden können.

(Um eine Python-Bibliothek zu installieren, die Sie zum Beispiel im Internet gefunden haben, folgen Sie den Anweisungen, aber verwenden sie `sage -python` anstelle von `python`. Oft bedeutet dies, dass `sage -python setup.py install` eingegeben werden muss.)

8.2 Ich möchte einen Beitrag zu Sage leisten. Wie kann ich dies tun?

Falls Sie für Sage einen Beitrag leisten möchten, wird Ihre Hilfe hoch geschätzt! Sie kann von wesentlichen Code-Beiträge bis zum Hinzufügen zur Sage-Dokumention oder zum Berichten von Fehlern reichen.

Schauen Sie sich die Sage-Webseite an um Informationen für Entwickler zu erhalten; neben anderen Dingen können Sie eine lange Liste nach Priorität und Kategorie geordneter, zu Sage gehörender Projekte finden. Auch der Sage Developer's Guide beinhaltet hilfreiche Informationen, und Sie können der `sage-devel` Google-Group beitreten.

8.3 Wie zitiere ich Sage?

Falls Sie ein Paper schreiben, das Sage verwendet, zitieren Sie bitte die Berechnungen die Sie mithilfe von Sage durchgeführt haben, indem Sie

```
[Sage] William A. Stein et al., Sage Mathematics Software (Version 4.3).
       The Sage Development Team, 2009, http://www.sagemath.org.
```

in Ihrem Literaturverzeichnis hinzufügen. (Ersetzen Sie hierbei 4.3 mit der von Ihnen benutzten Version von Sage.) Versuchen Sie bitte weiterhin festzustellen welche Komponenten von Sage in Ihrer Berechnung verwendet wurden, z.B. PARI?, GAP?, Singular? Maxima? und zitieren Sie diese Systeme ebenso. Falls Sie nicht sicher sind welche Software Ihre Berechnung verwendet, können Sie dies gerne in der `sage-devel` Google-Gruppe fragen. Lesen Sie *Polynome in einer Unbestimmten* um weitere Information darüber zu erhalten.

Falls Sie gerade das Tutorial vollständig durchgelesen haben, und noch wissen wie lange Sie hierfür gebraucht haben, lassen Sie und dies bitte in der `sage-devel` Google-Gruppe wissen.

Viel Spass mit Sage!

Anhang

9.1 Binäre arithmetische Operatorrangfolge

Was ist `3^2*4 + 2%5`? Der Wert (38) wird durch diese "Operatorrangfolge-Tabelle" festgelegt. Die Tabelle unterhalb basiert auf der Tabelle in Abschnitt § 5.15 des *Python Language Reference Manual* von G. Rossum und F. Drake. Die Operatoren sind hier in aufsteigender Ordnung der Bindungstärke aufgelistet.

Operatoren	Beschreibung
or	Boolesches oder
and	Boolesches und
not	Boolesches nicht
in, not in	Zugehörigkeit
is, is not	Identitätstest
>, <=, >, >=, ==, !=	Vergleich
+, -	Addition, Subtraktion
*, /, %	Multiplikation, Division, Restbildung
**, ^	Exponentiation

Um also `3^2*4 + 2%5` zu berechnen klammert Sage den Ausdruck in folgender Weise: `((3^2)*4) + (2%5)`. Es wird daher zuerst `3^2`, was `9` ist, dann wird sowohl `(3^2)*4` als auch `2%5` berechnet, und schließlich werden diese beiden Werte addiert.

Literaturverzeichnis

KAPITEL 11

Indizes und Tabellen

- *genindex*
- *modindex*
- *search*

Literaturverzeichnis

[Cyt] Cython, http://www.cython.org/.

[Dive] Dive into Python, online frei verfügbar unter http://diveintopython.org/.

[GAP] The GAP Group, GAP - Groups, Algorithms, and Programming, Version 4.4; 2005, http://www.gap-system.org/.

[GAPkg] GAP Packages, http://www.gap-system.org/Packages/packages.html/.

[GP] PARI/GP, http://pari.math.u-bordeaux.fr/.

[Ip] The IPython shell, http://ipython.scipy.org/.

[Jmol] Jmol: an open-source Java viewer for chemical structures in 3D, http://www.jmol.org/.

[Mag] Magma, http://magma.maths.usyd.edu.au/magma/.

[Max] Maxima, http://maxima.sf.net/.

[NagleEtAl2004] Nagle, Saff, and Snider. *Fundamentals of Differential Equations*. 6th edition, Addison-Wesley, 2004.

[Py] The Python language, http://www.python.org/ Reference Manual, http://docs.python.org/ref/ref.html/.

[PyDev] Guido, Some Guys, and a Mailing List: How Python is Developed, http://www.python.org/dev/dev_intro.html/.

[Pyr] Pyrex, http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/.

[PyT] The Python Tutorial, http://docs.python.org/tutorial/.

[SA] Sage web site, http://www.sagemath.org/.

[Si] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. Singular 3.3.1. A Computer Algebra System for Polynomial Computations. University of Kaiserslautern (2010), http://www.singular.uni-kl.de/.

[SJ] William Stein, David Joyner, Sage: System for Algebra and Geometry Experimentation, Comm. Computer Algebra {39}(2005)61-64.